启航计算机考研专业课系列

计算机考研专业课

数据结构



（考点详解 + 习题全解）

李红　刘财政　主编　　启航龙图计算机学院　审校

全面覆盖高频考点：
全书以408统考体系为主线，结合热门院校的自主命题

多方联合编写：
本书由高校在职教师、中科院博士与启航龙图教师团队共同编写

试题配有详细讲解：
所有考题配有详细步骤与解析，清晰易懂

强大的售后支持：
本书由启航龙图的专业教师团队为考生答疑解惑

清華大學出版社

启航计算机考研专业课系列

计算机考研专业课
数据结构一本通

（考点详解 + 习题全解）

李红　刘财政　主编

清華大学出版社
北　京

内 容 简 介

本书严格按照全国硕士研究生入学考试计算机学科专业基础综合大纲进行编写，内容涵盖线性表、树和二叉树、图、查找、排序等大纲要求的知识点，并以图表和代码的形式对考点进行讲解，注释清晰易懂。本书精选历年的 408 考题及部分名校试题进行详细讲解，帮助考生学练结合，提高考生的学习效率。

本书可作为学生参加计算机专业硕士研究生入学考试的辅导用书，也可作为计算机及相关专业的学生学习“数据结构”的教材。

图书在版编目（CIP）数据

计算机考研专业课. 数据结构一本通：考点详解+习题全解 / 李红，刘财政主编. —北京：清华大学出版社，2019
（启航计算机考研专业课系列）
ISBN 978-7-302-52708-4

I. ①计…　II. ①李…　②刘…　III. ①电子计算机－研究生－入学考试－题解　②数据结构－研究生－入学考试－题解　IV. ①TP3-44

中国版本图书馆 CIP 数据核字（2019）第 062358 号

责任编辑：袁金敏
封面设计：刘新新
责任校对：焦丽丽
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn, http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市铭诚印务有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　张：**22.25　　**字　数：**487 千字
（附习题全解 1 本）
版　　次：2019 年 7 月第 1 版　　**印　次：**2019 年 7 月第 1 次印刷
定　　价：69.00 元

产品编号：083244-01

前　言

“数据结构”是全国硕士研究生入学考试科目——计算机学科专业综合的考查科目之一，是研究“非数值计算的程序设计问题中计算机的操作对象及其关系和操作等的学科”，是介于数学、计算机硬件和计算机软件三者之间的一门核心课程，同时也是计算机科学与技术专业的专业基础课。课程内容不仅是一般程序设计（特别是非数值计算的程序设计）的基础，同时也是设计和实现编译程序、操作系统、数据库系统及其他系统程序和应用程序的重要基础。

“数据结构”主要研究数据及数据之间的关系、数据的存储以及具有关系的数据集上的操作。主要涉及三种关系：一对一关系（线性表）、一对多关系（树和二叉树）、多对多关系（图）。常见操作有创建、查找、删除、排序等。考生在复习时首先应熟练掌握理论内容，单纯的刷题无法解决基础内容的欠缺问题，也无法应对考场上不曾谋面的试题。然后再通过练习题及真题检验对课程知识的掌握程度。

本书分为两部分，第一部分是考点详解，第二部分为习题精讲。考点详解部分首先是导学部分，主要介绍本科目考试大纲和本书各章节知识点分布。第 1 章为绪论，主要介绍数据结构和算法的基本概念。第 2 章为线性表，主要介绍线性结构的基本概念、算法及实现，以及特殊线性表（栈和队列）、特殊矩阵等。第 3 章介绍树及二叉树的相关概念及算法。第 4 章介绍图的定义、算法。第 5 章介绍各种查找算法及其分析。第 6 章介绍排序算法及其特点。习题精讲部分主要收集全国硕士研究生入学考试计算机学科专业基础综合（专业课代码 408，本书文中统称 408）及各高校科研院所的历年真题，通过真题使考生零距离感受考题形式和答题思路，通过做题，熟练、灵活地掌握理论内容，进一步加强分析题目、求解问题的思维训练。

本书的知识详解部分尽可能多地给出基本操作的伪码实现、注释、算法的流程分析等，帮助考生深入理解算法本质，为解题打下坚实基础。此外，我们还为本书配备了丰富的视频讲解，扫描每章和图书封底的二维码即可观看。

备考过程中，考生需注意复习方法。首先应全局把握本书内容，熟悉本书特点、重点章节等。然后进行系统学习和总结，熟练掌握各知识点。最后通过历年真题分析巩固各知识点并了解各知识点的出题方式和考查频率。

备考过程漫长辛苦，考生应注意学习方法，提高复习效率，不搞消耗战，不做过多重复题，以不变应万变。毫无头绪时不妨归本还原，静下心来认真研究基本概念和算法，或许能打开解题思路。

编者

2019 年 1 月

目　录

第 0 章　导学

0.1　学 习 目 标


C 语言基础

要求考生比较系统地理解和掌握数据结构涉及的基本概念、原理和方法，能够综合运用所学原理和方法分析、判断、解决有关的理论问题和实际应用问题。

本书主要参考 408 考试大纲编写，学习目标具体如下。

（1）掌握数据结构的基本概念、基本原理和基本方法。

（2）掌握数据的逻辑结构、存储结构及基本操作的实现，能够对算法进行基本的时间复杂度与空间复杂度的分析。

（3）能够运用数据结构基本原理和方法进行问题的分析与求解，具备采用 C 或 C++ 语言设计与实现算法的能力。

0.2　大　　纲


常用算法思想

本书内容基于 408 考试大纲。

第 1 章绪论：基本概念，数据结构，算法。

第 2 章线性表：概述，线性表的存储，线性表的应用，栈、队列和数组，特殊矩阵的压缩存储。

第 3 章树与二叉树：树的基本概念，二叉树，树、森林，树与二叉树的应用。

第 4 章图：图的基本概念，图的存储及基本操作，图的遍历，最小（代价）生成树，关键路径。

第 5 章查找：查找的基本概念，顺序查找法，分块查找法，折半查找法，B 树及其基本操作、B+树的基本概念，散列（Hash）表，字符串模式匹配，查找算法的分析及应用。

第 6 章排序：排序的基本概念，插入排序，起泡排序，简单选择排序，希尔排序，快速排序，堆排序，二路归并排序，基数排序，外部排序，各种排序算法的比较，排序算法的应用。

0.3　本书知识结构

本书知识结构如图 0.1 所示。其中：加粗框中的内容需要重点理解并掌握。排序部

分仅列出重点内容——内部排序，其他见具体章节。

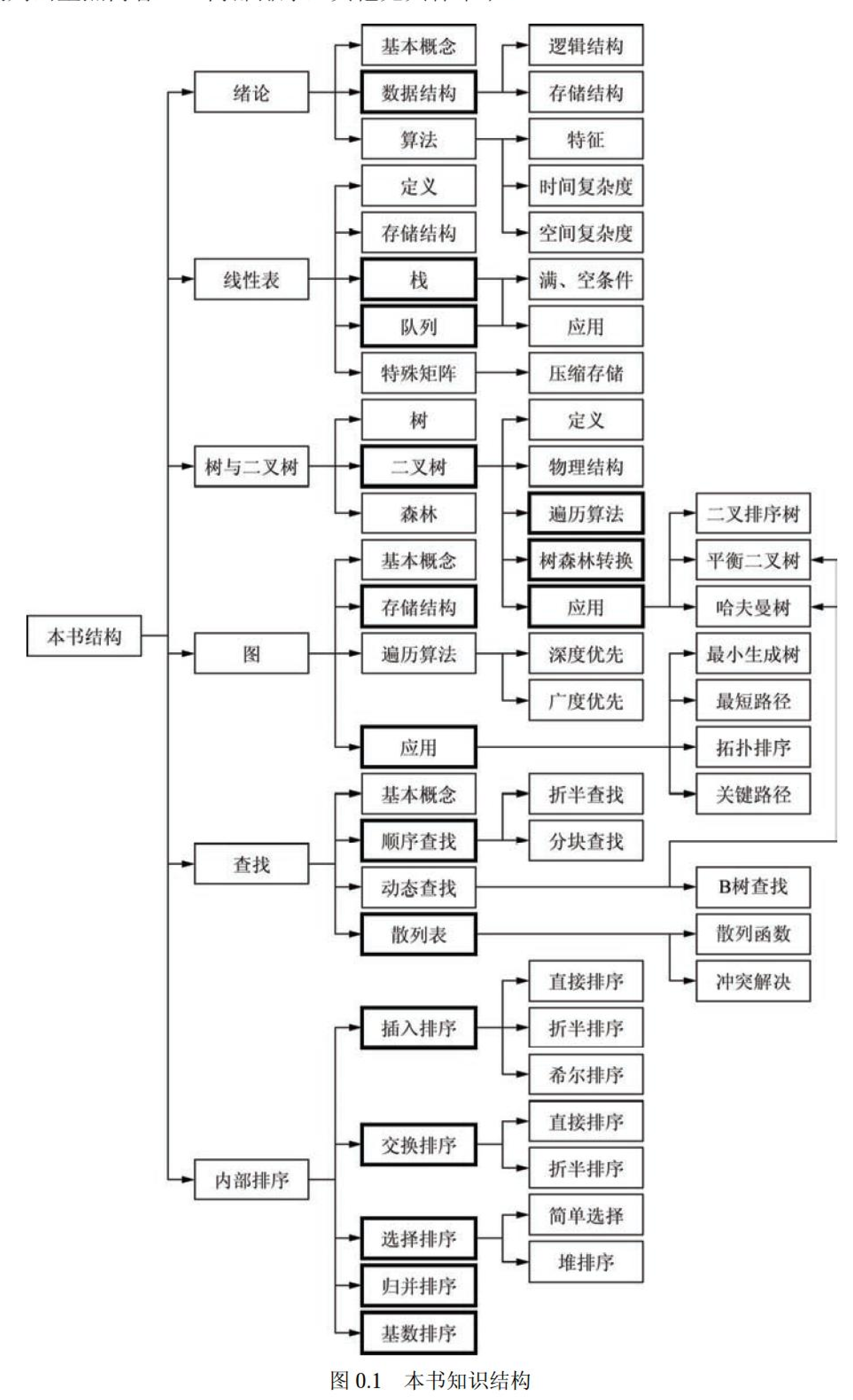


图 0.1　本书知识结构

第 1 章　绪论

本章学习目标

- 了解数据、数据元素、数据项等基本概念。
- 深入理解数据结构的含义、逻辑结构和存储结构的概念。
- 理解不同逻辑结构的本质区别。
- 掌握不同存储结构的实质。
- 理解算法的概念。
- 掌握算法的特征。
- 能够分析各种算法的时间复杂度和空间复杂度。

1.1 本章导学



绪论

1.1.1 知识结构

本章知识结构如图 1.1 所示，加粗框中的内容需要重点理解并掌握。

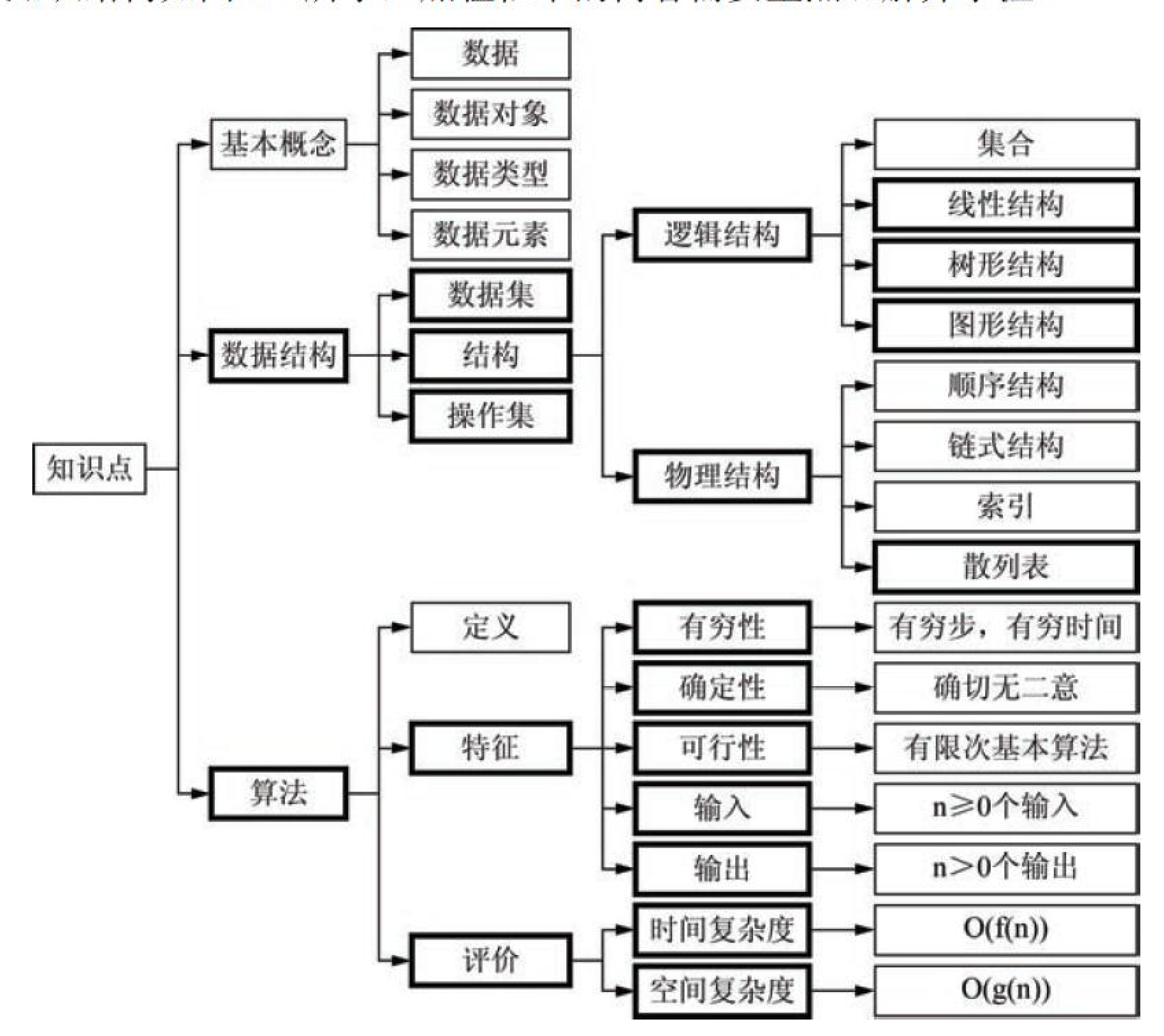


图 1.1　本章知识结构

1.1.2 命题特点

1. 命题规律

（1）本章是历年各高校硕士研究生招生考试的基本考查内容，命题形式既有客观题也有主观题。

（2）本章既可单独命题，也可与后续章节联合命题。

（3）顺序存储结构、链式存储结构的操作细节易出客观题。

（4）链表操作的综合应用易出主观题。

2. 考查趋势

本章在全国硕士研究生入学考试中的重要性近年不会改变，主观题型、客观题型出现的概率极大。特别注意顺序存储结构与查找、排序两章联合命题，以及链式存储结构的综合应用。

1.2 基本概念

（1）数据：客观事物的符号表示，指所有能输入到计算机中并被计算机程序处理的符号。

（2）数据元素：又称结点，是数据的基本单位，在计算机中通常作为一个整体进行处理。该概念根据具体问题进行具体界定，外延可大可小。

（3）数据项：又称属性，指数据中具有独立意义的、不可分割的最小单位。

注意，数据由多个数据元素组成，一个数据元素又包含若干数据项。

（4）数据对象：性质相同的数据元素的集合。

（5）数据类型：一组性质相同的值集合以及定义在该集合上的一组操作的总称。

（6）抽象数据类型：Abstract Data Type，简称 ADT，指一个数学模型及定义在该模型上的一组操作。

（7）逻辑结构：数据元素之间固有的逻辑关系。该关系与计算机无关，是人的思维层面对现实世界数据之间关系的理解。

（8）存储结构：逻辑结构在计算机中的表示，也称物理结构，不同存储结构对数据处理效率具有较大影响。

（9）数据结构：相互之间具有特定关系的数据元素的集合，包含数据集、结构（逻辑结构、物理结构）及施加其上的操作集。

（10）算法：解决特定问题的有限指令序列。

1.3　数 据 结 构

1.3.1　定义

数据结构指相互之间存在一种或多种关系的数据元素的集合。数据元素不是孤立存在的，它们之间存在着这样或那样的关系，数据元素之间的关系称为结构，包含逻辑结构和存储结构两个层面，以及数据集上的一组操作。

其中，逻辑结构是数据结构的抽象，存储结构是数据结构的实现，两者综合起来建立数据元素之间的结构关系。

数据结构注重数据元素之间的相互关系与组织方式、运算及规则，不涉及数据元素的具体内容。

数据结构的形式化定义有如下三种常见形式。

（1）二元组：Data_Structures = (D, R)，其中，D 是数据元素的有限集，R 是 D 上关系的有限集。

【例 1-1】 有一种数据结构 A=(D,R)，其中：

D={1,2,3,4,5,6,7,8,9,10}，R={<1,2>, <2,3>, <3,4>, <4,5>, <5,6> , <6,7>, <7,8>, <8,9>, <9,10>}。

（2）图形：用○表示一个元素，用横线或箭头连接两个○表示元素之间的关系，例如：

①→②→③→④→⑤→⑥→⑦→⑧→⑨→⑩

（3）抽象数据类型：用三元组(D, S, P)表示，其中 D 是数据对象的集合，S 是 D 上关系的集合，P 是 D 的基本操作集合，形式如下。

```
ADT 抽象数据类型名 {
    数据对象：〈数据对象的定义〉
    数据关系：〈数据关系的定义〉
    基本操作：〈基本操作的定义〉
} ADT 抽象数据类型名;
```

例如：抽象数据类型：矩形

① 矩形 ADT 的定义。

```
ADT Rectangle {
数据对象：  length;    // 非负实数，矩形的长
            width;     // 非负实数，矩形的宽
数据关系：  无
基本操作：
            init( &R, length, width );  // 将矩形 R 的长和宽分别初始化为 length 和
                                        // width
            area(R);                    // 返回矩形 R 的面积
            circumference(R);           // 返回矩形 R 的周长
} ADT Rectangle;
```

② 矩形 ADT 的表示。

```
// 定义矩形的存储结构
typedef struct
{
    float length;        // 矩形的长
    float width;         // 矩形的宽
} Rectangle;
// 定义矩形的基本操作
bool  init(Rectangle &R, float l, float w);
float  area(Rectangle R);
float  circumference(Rectangle R);
```

③ 矩形 ADT 的实现。

```
bool init(Rectangle &R, float l, float w)
{
    if( l>0 && w>0 ){
        R.length=l;
        R.width=w;
        return true;
    }
    else
        return false;
}
float area(Rectangle R)
{
    return R.length*R.width;
}
float circumference(Rectangle R)
{
    return 2*(R.length+R.width);
}
```

抽象数据类型是近年来计算机科学领域提出的最重要概念之一，集中体现了程序设计的一些最基本的原则，其特点具体如下。

- 数据抽象与信息隐藏。
    - 一个抽象数据类型确定了一个数学模型，并将模型的实现细节加以隐藏。
    - 定义了一组运算，并将运算的实现过程隐藏起来。
- 模块化。
- 继承性。
- 封装与复用。

1.3.2 逻辑结构

数据元素之间的逻辑关系，可以用一个数据元素的集合和定义在此集合上的若干关系来表示。根据数据元素之间逻辑关系的不同，数据元素之间的逻辑结构一般分为以下4种基本类型。

1. 集合

数据元素之间最松散的一种结构，仅具有属于同一集合这种关系，集合中无重复元

素，如图 1.2 所示。本书不讨论这种逻辑结构。

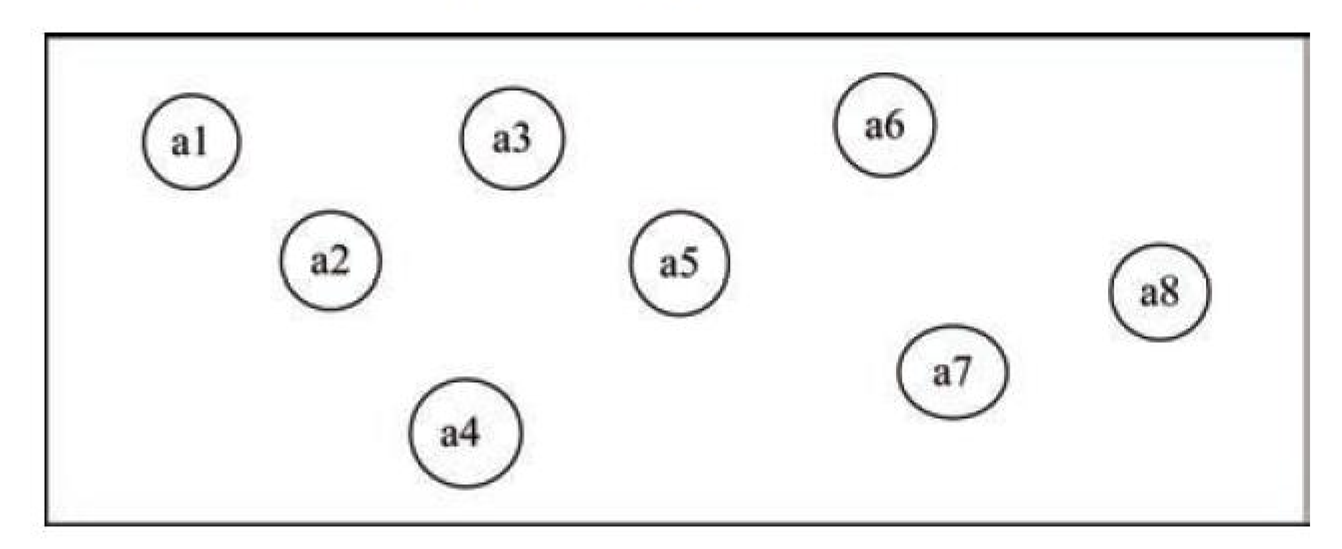


图 1.2　集合

2．线性结构

数据元素之间具有 1∶1 关系，是简单、常见的逻辑结构，如图 1.3 所示。

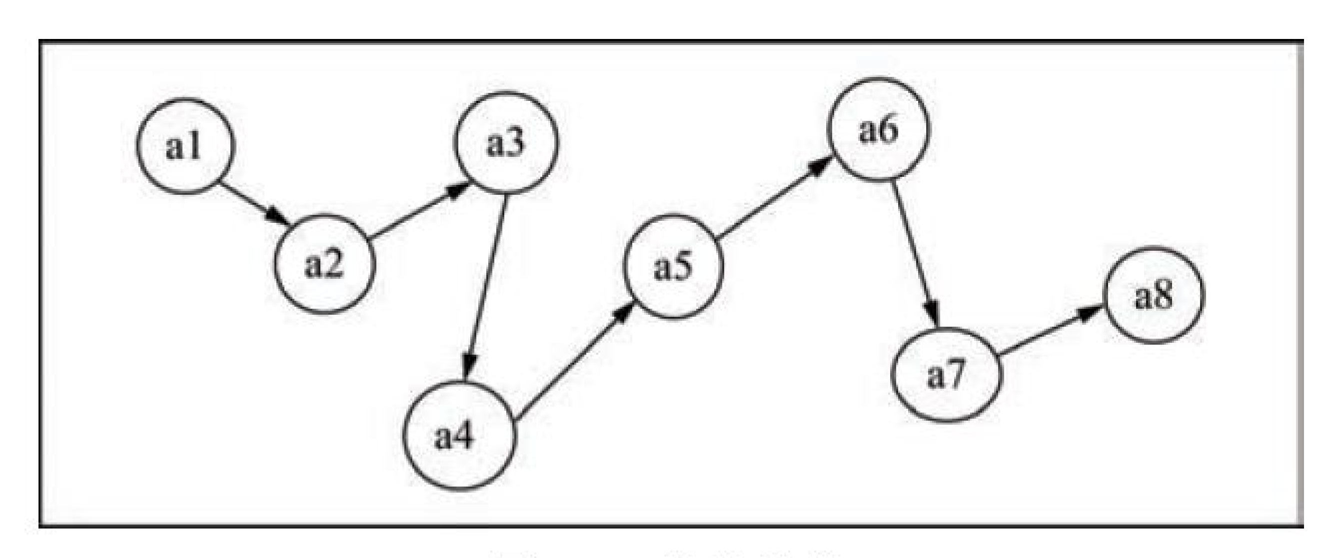


图 1.3　线性结构

3．树形结构

数据元素之间具有 1∶n 关系，为常见逻辑结构，如图 1.4 所示。

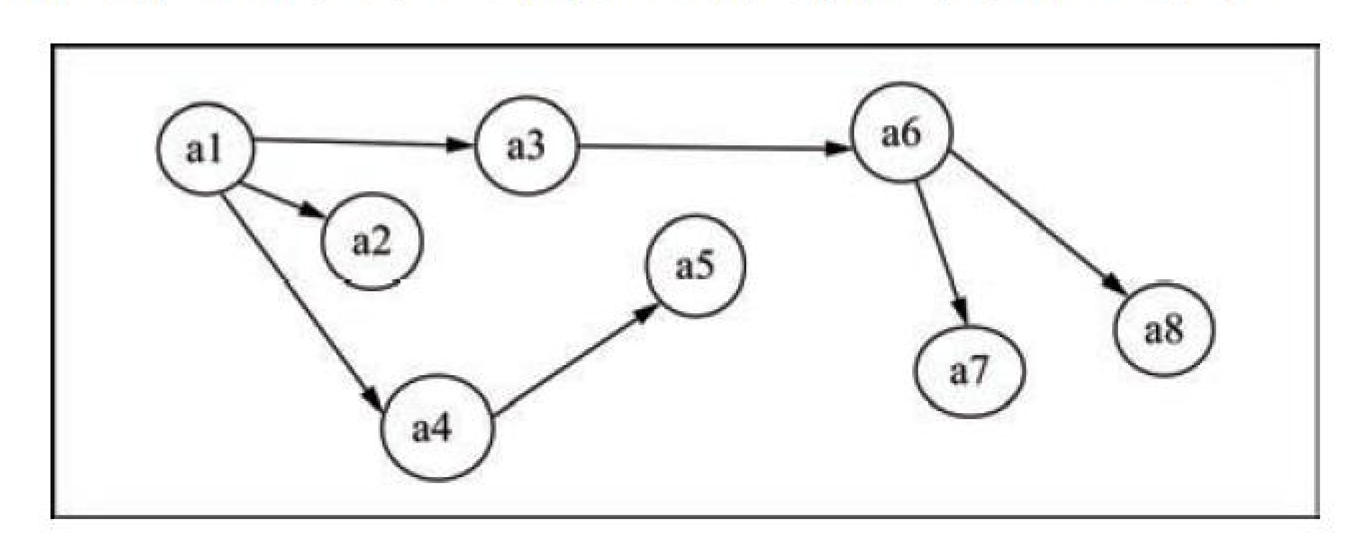


图 1.4　树形结构

4．图形结构

数据元素之间具有 m∶n 关系，为常见逻辑结构，如图 1.5 所示。

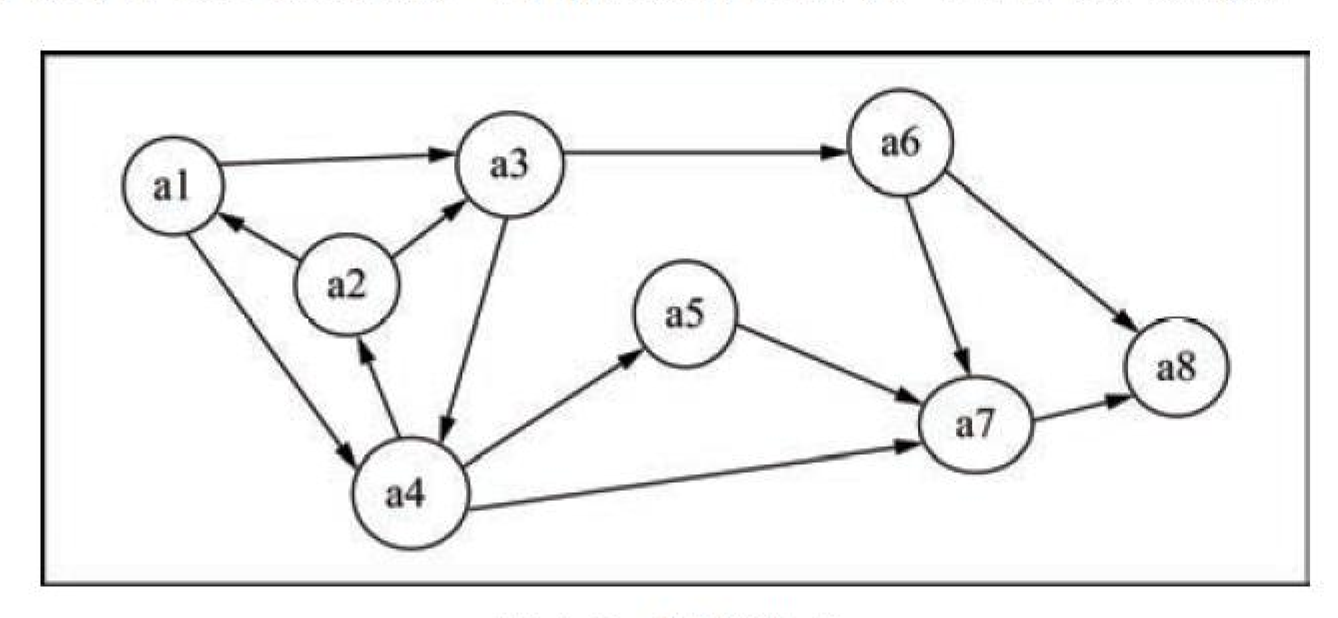


图 1.5　图形结构

线性结构为特殊树形结构，树形结构为特殊图形结构。

1.3.3　存储结构

存储结构是逻辑结构在计算机中的表示和实现。根据数据元素在计算机内部的存储方式不同，数据元素在内存中的物理结构一般分为以下 4 种基本类型。

1．顺序结构

逻辑结构中相邻的数据元素存储在计算机连续的内存单元中，即逻辑关系通过存储单元之间的相对位置表示。图 1.3 的线性结构对应的顺序存储结构如图 1.6 所示。

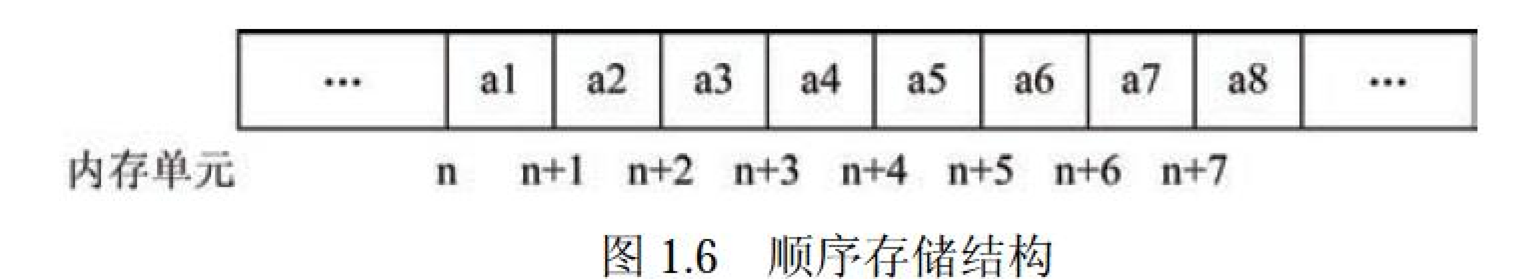


图 1.6　顺序存储结构

此结构的优点包括：

- 不需要额外空间存储逻辑关系，节省存储空间。
- 支持随机访问。

此结构的缺点包括：

- 为确保顺序存储方式的特性，进行数据元素的插入和删除操作需要移动部分甚至全部数据元素，改变其存储位置。
- 存储所有数据元素需要连续的存储空间。

2．链式结构

采用链式结构，逻辑结构中相邻的数据元素可以存储在计算机中不连续的内存单元中，逻辑关系通过数据元素中附加的指针域表示。图 1.3 的线性结构对应的链式存储结构如图 1.7 所示。

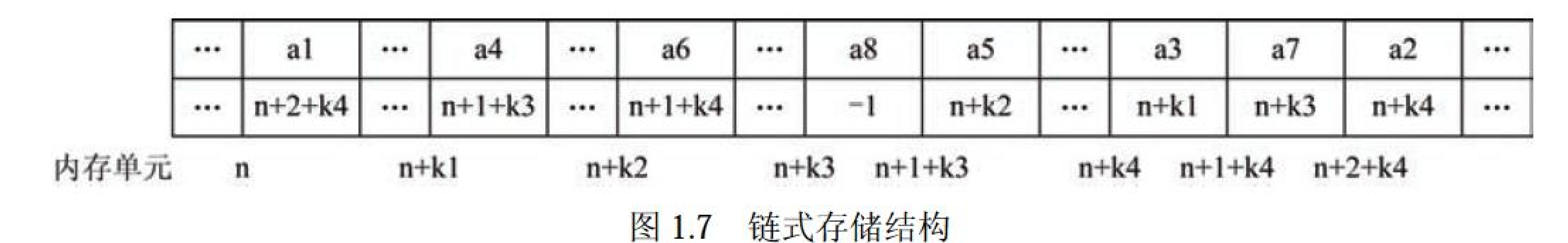


图 1.7　链式存储结构

head=n;　// head 为头指针，表示第一个元素的存储位置

此结构的优点包括：

- 进行数据元素的插入和删除操作仅需修改相关数据元素的指针域，无须移动其他数据元素，无须改变其存储位置。
- 修改数据效率高。
- 存储所有数据元素不需要连续的存储空间。

此结构的缺点包括：

- 需要额外空间——通过指针域存储逻辑关系，单个数据元素和整体数据的存储空

间增大。

- 不支持随机访问。

3. 索引结构

索引结构通过索引表指示数据元素的存储位置，索引表由索引项构成。索引项的一般形式为（关键字，地址），其中关键字是能够唯一标识数据元素的数据项，地址表示数据元素在内存的存储位置。图 1.3 的线性结构对应的索引存储结构如图 1.8 所示，表 1.1 为相应的索引表，ai.key 为 ai 的关键字，i∈[1,8]。

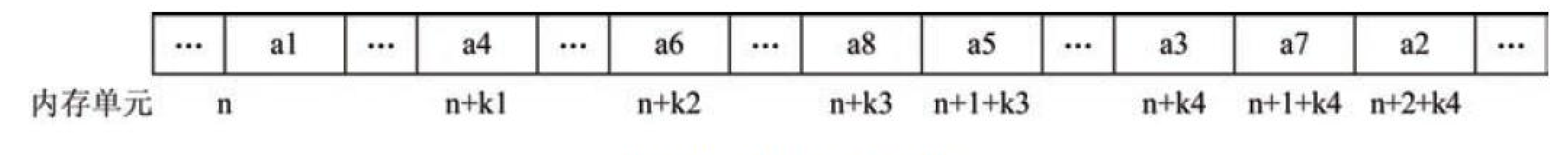


图 1.8　索引存储结构

表 1.1　索引表

关键字	地址	关键字	地址
a1.key	n	a5.key	n+1+k3
a2.key	n+2+k4	a6.key	n+ k2
a3.key	n+k4	a7.key	n+1+k4
a4.key	n+k1	a8.key	n+k3

此结构的优点包括：

- 进行数据元素的插入和删除操作仅需修改索引表中相关数据元素的存储地址，无须移动数据元素。
- 修改数据效率高。
- 支持随机访问。
- 存储所有数据元素不需要连续的存储空间。

此结构的缺点包括：

- 需要额外存储空间——通过索引表存储逻辑关系。
- 需要额外时间——对索引表进行维护。

4. 散列结构

数据元素的存储单元地址由散列（Hash）函数根据其关键字计算得出，后续章节会有详细介绍。图 1.3 的线性结构对应的散列存储结构如图 1.9 所示，假设 ai.key 为 ai 的关键字，a1.key=5，a2.key=15，a3.key=9，a4.key=20，a5.key=6，a6.key=16，a7.key=18，a8.key=12，散列函数为 H(ai.key)= ai.key % 18，%为取余运算。则：

H(a1.key)= 5 % 18=5　　　H(a2.key)= 15 % 18=15

H(a3.key)= 9 % 18=9　　　H(a4.key)= 20 % 18=2

H(a5.key)= 6 % 18=6　　　H(a6.key)= 16 % 18=16

H(a7.key)= 18 % 18=0　　　H(a8.key)= 12 % 18=12

存储如下：

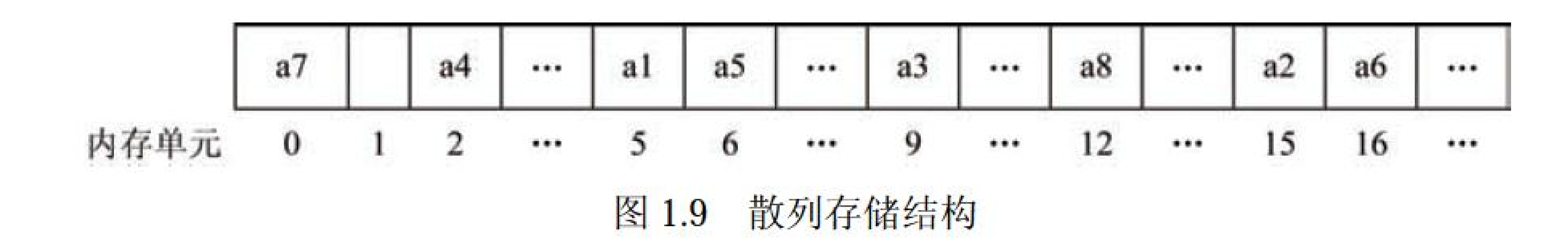


图 1.9　散列存储结构

此结构的优点包括：

- 进行数据元素的插入和删除操作仅需通过散列函数根据关键字计算该数据元素的存储位置，无须移动数据元素。
- 查找速度快。
- 存储所有数据元素不需要连续的存储空间。

此结构的缺点包括：

- 需要好的散列函数——数据元素计算得到的存储地址尽可能“散”，无重复。
- 计算得到的不同数据元素的存储地址难以完全避免冲突，所以需要有效的处理冲突的方法。

1.4 算　　法

1.4.1 定义

通俗讲，算法是一种解题方法，是解决某特定现实问题的一个过程或一种策略。

严格讲，算法是对特定问题求解步骤的一种描述，是一个有穷的规则集合，这些规则为解决某一特定任务规定了一个有序的运算序列。也可以说算法是指令的有限序列，其中每一条指令表示一个或多个操作。

计算机对数据的处理可分为数值数据处理和非数值数据处理两种，其中数值处理主要进行算术运算，非数值处理主要进行查找、排序、插入、删除等运算。

描述算法的主要方法有自然语言、程序设计语言（或类程序设计语言）、流程图（包括传统流程图和 N-S 结构图）、伪语言和 PAD 图。

1.4.2 特征

算法必须满足以下五个重要特性。

（1）有穷性：一个算法必须能在执行有穷步之后正常结束，且每一步都在有穷时间内完成，即算法必须在有限时间内完成，不能无穷循环。

（2）确定性：算法的每一步必须有确切的含义，无二义性，即输入相同数据必须得到同样的输出结果。

（3）可行性：算法描述的所有操作均可通过执行有限次已实现的基本运算实现。

（4）输入：一个算法应该具有 0 个或多个输入，这些输入取自某特定的数据对象集合。

（5）输出：一个算法应该具有一个或多个输出，这些输出和输入之间存在某种特定的关系。

1.4.3　算法和程序

算法是对特定问题求解步骤的一种描述，与所用计算机和所选编程语言无关；程序是算法在计算机中的实现，与所用计算机和所选编程语言有关。

程序=算法+数据结构。

程序不一定满足有穷性（如操作系统在用户未使用前一直处于踏步等待状态，直到新的用户事件出现，而算法必须满足有穷性。

程序中的指令必须是机器可执行的，不能有语法错，而算法则无此限制，算法可以通过程序表述，而程序不能用算法代替。

算法和程序都是表达解决问题方法的逻辑步骤，但算法独立于具体的计算机，与具体的编程语言无关，而程序正好相反。

程序是算法，但算法不一定是程序。

1.4.4　评价

一个好算法一般应该具有如下特点。

（1）正确性：对于任何输入数据都能得出满足要求的结果。

（2）可读性：易于阅读和理解。

（3）健壮性：对非法输入的恰当处理，使系统能在非法输入下给出适当反馈或正常退出。

（4）高效性：时间效率高，空间消耗少。

其中需要重点分析高效性，即在问题规模 n 下对算法运行时所花费的时间 T、所占用的存储空间 S 的评价。

1. 时间复杂度

（1）时间复杂度是算法对应程序在机器中运执行时所需的时间。

算法的执行时间=Σ算法中基本操作执行次数×该基本操作执行时间。

算法的时间复杂度计算相当烦琐，一般没必要精确地计算出算法的时间复杂度，只要大致计算出相应的数量级 (Order) 即可，$T(n)=O(f(n))$。

（2）时间复杂度的影响因素包括如下几点：

① 问题规模。

② 算法实现所用的编程语言。

③ 编译程序生成的目标代码质量。

④ 硬件速度。

（3）常见时间复杂度包括：

① 常量阶：$T(n)=O(1)$。

```
x=x+1;
```

② 线性阶：$T(n)=O(n)$。

```
for (i=1; i<= n; i++)
   x=x+1;
```

③ 平方阶：$T(n)=O(n^2)$。

```
for (i=1; i<= n; i++)
   for (j=1;j<= n; j++)
      x=x+1;
for (i=0; i<n-1; i++)
   for (j=i+1; j<n; j++)
      if (a[i] < a[j])
         m=a[i], a[i]=a[j], a[j]=m;
```

时间复杂度以最大语句频度 if 语句的频度 n^2 来估算，不考虑算法中其他语句频度：

$$(n-1)+(n-2)+\cdots+1=n(n-1)/2=(n^2-n)/2$$

$$T(n)=O(n^2)$$

④ 立方阶：$T(n)=O(n^3)$。

```
for (i=1; i<= n; i++)
   for (j=1; j<= n; j++)
      for (k=1; k<= n; k++)
         n++;
```

（4）不同时间复杂度的比较。

① 多项式级：$O(1)<O(n)$。

② 对数级：$O(\log_2 n)<O(n\log_2 n)$。

③ 平方立方级：$O(n^2)<O(n^3)$。

④ 指数级：$O(2^n)<O(n^n)$。

$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)<O(n^n)$。

指数级的时间复杂度非常高，多需转换为多项式级时间复杂度。

2．空间复杂度

空间复杂度是算法对应程序在机器执行过程中所占用的临时存储空间，包含三部分：

（1）算法本身所占用的存储空间。

（2）输入数据所占用的存储空间。

（3）算法在运行过程中临时占用的存储空间。

其中前两部分和算法无关，所以算法的空间复杂度一般仅考虑第三部分。

与时间复杂度一样，空间复杂度也用数量级（Order）表示，即 $S(n)=O(g(n))$。

1.5 本章小结

本章主要介绍数据结构的基本概念，重点如下：

（1）理解常见逻辑结构的概念和区别。

（2）深入理解不同存储结构的特点及适用场合。

（3）熟练掌握算法的概念、特征和评价方法。

第 2 章　线性表

本章学习目标

- 理解线性表的基本概念和特点。
- 掌握两类存储结构的本质。
- 熟练掌握两类存储结构的操作细节，尤其是单链表、单循环链表、双链表、双循环链表在操作上的差异。
- 掌握本章内容和后续查找、排序等章节的结合出题。

2.1 本 章 导 学



线性表

2.1.1 知识结构

本章知识结构如图 2.1 所示，加粗框中的内容需要考生重点理解并掌握。

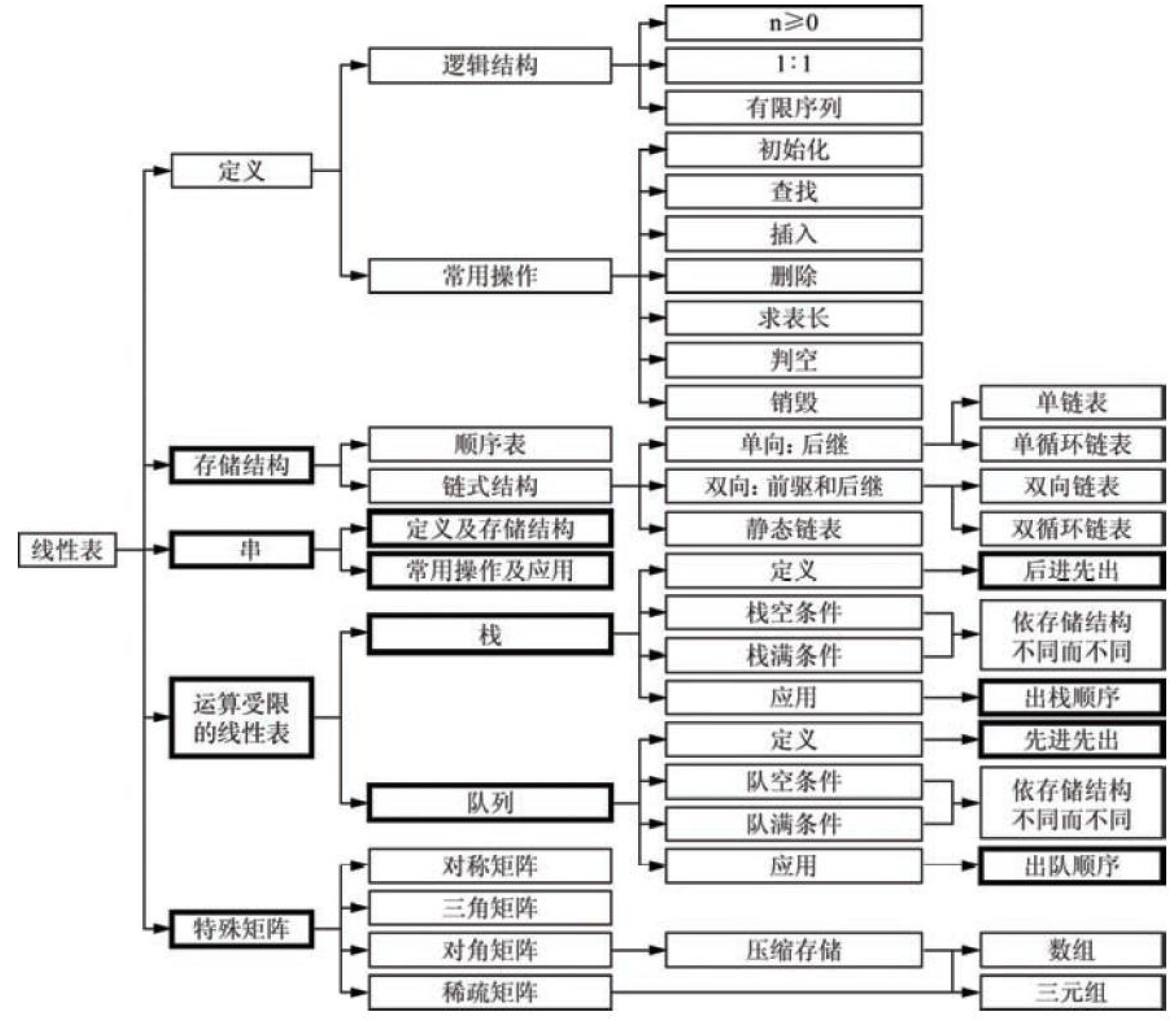


图 2.1　本章知识结构

2.1.2　命题特点

1．命题规律

（1）本章为各高校硕士研究生招生考试的重点考查内容，既有客观题又有主观题。

（2）本章内容既可单独命题，也可与后续章节联合命题。

（3）顺序存储结构、链式存储结构的操作细节易出客观题，链表操作的综合应用易出主观题。

2．考核趋势

本章在各高校硕士研究生入学考试中占据重要地位，其基本概念和原理对于后续章节具有重要参考作用，主观、客观题目均易出。

考生需特别注意以下内容。

（1）顺序存储结构与查找、排序两章联合命题。

（2）链式存储结构的综合应用。

（3）栈、队列的特点及应用。

（4）串的特点、存储及应用。

（5）特殊矩阵的存储及操作。

2.2　线性表概述

线性表是最基本、最常用的数据结构，表中各元素的逻辑关系具有 1∶1 的串行特性，编程实现时根据实际应用场景可以采用不同的存储结构，使逻辑上相邻的元素对应的存储位置未必相邻，增加了应用的灵活性。

2.2.1　定义

1．概念

具有相同数据类型的 n（n≥0）个数据元素的有限、有序序列称为线性表，n 为表长，n=0 时称为空表。

以下 2 点需特别注意：

（1）“有限”指的是任何线性表包含的元素个数是有限的。

（2）“有序”指的是逻辑上的先后顺序，而非存储空间的前后位置。

2．非空线性表的特点

（1）同一线性表中的所有数据元素具有相同的数据类型。

（2）线性表中的数据元素个数有限；

（3）数据元素之间总体满足 1∶1 的逻辑关系。

（4）存在唯一被称为“第一个”的元素（表头），该元素只有后继，没有前驱。

（5）存在唯一被称为“最后一个”的元素（表尾），该元素只有前驱，没有后继。

（6）除第一个元素外，其他元素均只有唯一的前驱。

（7）除最后一个元素外，其他元素均只有唯一的后继。

（8）线性表的抽象数据类型 ADT 描述

```
ADT list{
    数据对象 D：D={ a_i| a_i|∈eleSet, i=1,2,…,n, n≥0}
    数据关系 R：R={< a_{i-1}, a_i > | | a_{i-1}, a_i∈D, i=2,…,n, n≥0}
    基本操作 P：
      void initList(*L);              // 初始化线性表，构造空线性表，表长为 0
      void insertList (*L,i,e);       // 在线性表的第 i 个元素之前插入一个元素 e
      unsigned  listLength(L);        // 求线性表的长度
      eleType  getElem1(L,i);         // 获取线性表的第 i 个元素
      void getElem2(L,i,*e);          // 获取线性表的第 i 个元素，放入 e
      unsigned  locateElem(L,e);      // 返回 e 在线性表中的位置
      void listDelete(*L,i,*e);       // 删除线性表的第 i 个元素,将其值存放于变量 e
      void printList (L,visit());     // 遍历线性表
      int  emptyList (L);             // 线性表判空
      void destroyList(*L);           // 销毁线性表
} ADT list
```

3．应用场合

线性表是最简单、最常用的数据结构。对于一个实际应用问题，如果待处理的数据元素性质相同，且任意数据元素之间具有串行关系，就可以将其抽象为线性表。例如学生成绩、销售数据等。

2.2.2 基本操作

基本操作指常用、必要的操作，其他操作可以由基本操作实现。同一基本操作在不同存储结构下的实现方法不同。

（1）初始化线性表：initList(*L)，构造空线性表，表长为 0，但顺序表的初始化还需为线性表申请到初始存储空间。返回值——新构建的线性表由地址形式的形参带回调用函数，函数返回值类型为空类型 void。

（2）在线性表的第 i 个元素之前插入一个元素 e： insertList (*L,i,e)，对线性表的改变通过地址形式的形参 L 返回，函数返回值类型为空类型 void。

（3）求线性表的长度：listLength(L)，由函数中的返回语句返回线性表当前的数据元素个数，形参为值参，函数返回值类型为整型 int 或无符号类型 unsigned。

（4）获取线性表的第 i 个元素： 即按照逻辑位置查找线性表 L 的第 i 个数据元素，返回值可通过函数中的返回语句返回，比如 getElem1(L,i)，此时函数返回值类型应该和数据元素类型一致；也可通过地址形式的形参带回调用函数，比如 getElem2(L,i,*e)，此时函数返回值类型为空类型 void。

（5）确定元素 e 是线性表的第几个元素：locateElem(L,e)，即按值查找线性表中是否存在值为 e 的数据元素。若存在，返回其逻辑位置，否则返回 0。函数返回值类型为无符号类型 unsigned。

（6）删除线性表的第 i 个元素，并将其值存放于变量 e 中：listDelete(*L,i,*e)，对线性表的改变通过地址形式的形参 L 和 e 返回，函数返回值类型为空类型 void。

（7）遍历线性表：printList(L,visit())，以 visit()方式输出线性表 L 中的所有元素，函数返回值类型为空类型 void。

（8）线性表判空：emptyList (L)，判断线性表 L 中的数据元素个数是否为 0，或者表长是否为 0，函数返回值类型为布尔型，C 语言中可用整型 int 替代，若线性表为空，返回非零值（真），否则返回 0（假）。

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，函数返回值类型为空类型 void。

2.3　线性表存储结构及操作实现

2.3.1　顺序表

1．概念

顺序表即线性表的顺序存储方式，通常用一组地址连续的内存单元（数组）依次存储线性表的所有元素，即逻辑上相邻的数据元素存储在相邻的物理空间中，物理位置关系反映了逻辑关系。

2．存储形式

假设某线性表为 $L=\{D,R\}$，$D=\{a_i\}$，$i\in[1,n]$，$R=\{<a_{i-1},a_i>\}$，其中 a_{i-1} 称为 a_i 的直接前驱，a_i 称为 a_{i-1} 的直接后继，a_0 无直接前驱，a_n 无直接后继。假设每个数据元素占 1 个单元，则其顺序表存储形式如图 2.2 所示。

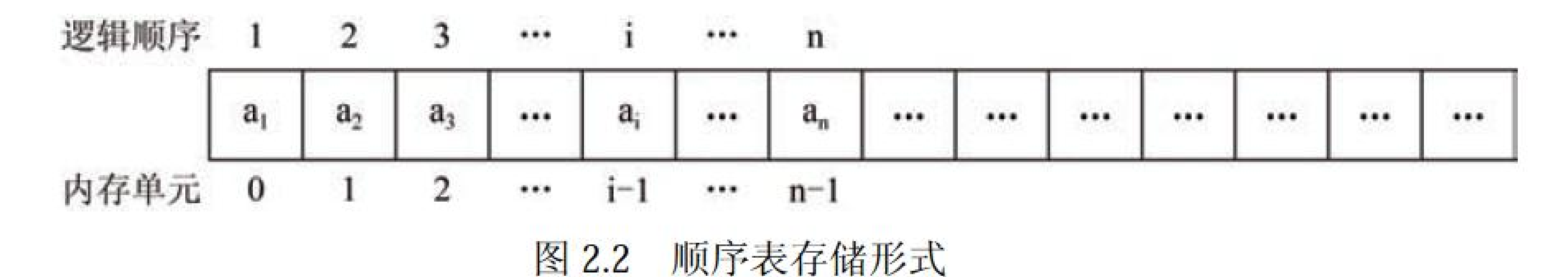


图 2.2　顺序表存储形式

3．定义

顺序表的定义以数组为主体，按照数组大小是否可变分为静态分配和动态分配两种形式。假设数据元素的类型为 eleType。

1）静态分配

编译时已确定数组的大小和位置，程序运行期间不可改变。

```
#define maxSize 1000          // 顺序表的最大长度
typedef  struct {             // 定义结构体类型
   eleType Selem[maxSize];   // 线性表存储于数组 Selem，一次性申请 maxSize 个数
                              // 据元素所需的存储空间 maxSize*sizeof(eleType)
                              // 个存储单元
  unsigned  length;           // 顺序表的当前长度，增删元素时需同步进行加减操作，
                              // length 取值范围为 0～maxSize-1
}Slist;                       // 静态分配空间的顺序表的类型名为 Slist
```

注意：

- 若 length 已等于 maxSize，则不可进行插入操作。
- 若 length 已等于 0，则不可进行删除操作。

2）动态分配

初始化程序执行期间通过 malloc 函数为数组申请空间，程序运行期间若空间不够，可通过 realloc 函数在保留原存储值的前提下为数组重新申请更大的连续存储空间。

```
#define initSize 1000     // 顺序表的初始长度
#define incSize 500       // 增大顺序表存储空间时，每次的增长值
typedef  struct {         // 定义结构体类型
   eleType *Delem;        // 线性表存储于指向数组的指针 Delem，当前该数组并不存
                          // 在，程序运行期间需要申请 initSize*sizeof(eleType)
                          // 个存储单元
   unsigned  length;      // 顺序表的当前长度，增删元素时需同步进行加减操作，
                          // length 取值为不超过当前申请存储单元个数的无符号数
}Dlist;                   // 动态分配空间的顺序表的类型名为 Dlist
```

注意：

- 初始化时，为顺序表申请 initSize 个数据元素所需的连续存储空间，首地址存放于指针变量 Delem。
- 若 length 已等于 maxSize，进行插入操作前需执行 realloc 函数，这样既保留原存储值，又能为数组重新申请更大的连续存储空间，每次增长 500 个数据元素所占的存储空间量。
- 若 length 已等于 0，则不可进行删除操作。

4. 基本操作实现

1）初始化线性表：initList(*L)

（1）静态分配代码如下。

```
void initList(SList *L){
    L->Selem=(eleType *)malloc(maxSize *sizeof(eleType));
    if(!L->Selem)                    // 没有分配成功
       exit(OVERFLOW);               // 退出程序，提示溢出
    L->length=0;
    return;
}
```

（2）动态分配代码如下。

```
void initList(DList *L){
     L->Delem=(eleType *)malloc(initSize *sizeof(eleType));
     if(!L->Delem)                        // 没有分配成功
        exit(OVERFLOW);                   // 退出程序，提示溢出
     L->length=0;
     return;
}
```

2）在线性表 L 的第 i 个元素(逻辑位置)之前插入一个元素 e：insertList (*L,i,e)

（1）静态分配代码如下。

```
void insertList(SList *L,unsigned i, eleType e){
    if(L->length==maxSize)               // 存储空间已满
       exit(OVERFLOW);                   // 退出程序，提示溢出
    if(i<1 || i>L->length)               // 非法逻辑位置
       exit(ERROR);                      // 退出程序，提示位置出错
    for(unsigned j=L->length-1;j>=i-1;j--)
        L->Selem[j+1]= L->Selem[j]; // 从表尾开始到插入位置，数据元素依次后
                                         // 移一个位置
    L->Selem[i-1]=e;                     // e 插入到线性表的第 i 个位置
    L->length++;                         // 表长加 1
    return;
}
```

（2）动态分配代码如下。

```
void insertList(DList *L,unsigned i, eleType e){
    if(L->length==initSize){             // 存储空间已满
      eleType *p;
      p=(eleType*)realloc(L->Delem,(initSize+incSize) *sizeof(eleType));
                                         // 重新申请(initSize+incSize)*sizeof-
                                         // (eleType) 大小的存储空间，L->Delem
                                         // 中的 L->length 个数据元素复制过来，新
                                         // 空间首地址为 p
       if(!p)                            // 没有分配成功
          exit(OVERFLOW);                // 退出程序，提示溢出
       L->Delem=p;                       // L->Delem 指向新申请到的存储空间
      L->length+= incSize;               // 表长修改为新的存储空间可存放数据元
                                         // 素个数
    }
    if(i<1 || i>L->length)               // 非法逻辑位置
       exit(ERROR);                      // 退出程序，提示位置出错
    for(unsigned j=L->length-1;j>=i-1;j--)
        L->Delem[j+1]= L->Delem[j]; // 从表尾开始到插入位置，数据元素依次后
                                         // 移一个位置
    L->Delem[i-1]=e;                     // e 插入到线性表的第 i 个位置
    return;
}
```

3）求线性表的长度：listLength(L)

（1）静态分配代码如下。

```
    unsigned  listLength (SList L){
            return  L.length;
    }
```

（2）动态分配代码如下。

```
   unsigned  listLength(DList L){
             return  L.length;
   }
```

4）获取线性表的第 i 个元素：getElem1(L,i)或 getElem2(L,i,*e)

（1）静态分配代码如下。

```
eleType  getElem1(SList L,unsigned i) {  //i 为逻辑位置,取值范围为 1～L.length
    if(i<1  || i>L.length)                // 非法逻辑位置
       exit(ERROR);                       // 退出程序，提示位置出错
     return L.Selem[i];
}
```

或

```
void getElem2(SList L,unsigned i, eleType *e) {
                                          //i 为逻辑位置,取值范围为 1～L.length
    if(i<1  || i>L.length)                // 非法逻辑位置
       exit(ERROR);                       // 退出程序，提示位置出错
     *e=L.Selem[i];
     return;
}
```

（2）动态分配代码如下。

```
eleType  getElem1(DList L,unsigned i) {  //i 为逻辑位置,取值范围为 1～L.length
    if(i<1  || i>L.length)                // 非法逻辑位置
       exit(ERROR);                       // 退出程序，提示位置出错
     return L.Delem[i];
}
```

或

```
void getElem2(DList L,unsigned i, eleType *e) {
                                          //i 为逻辑位置,取值范围为 1～L.length
    if(i<1  || i>L.length)                // 非法逻辑位置
       exit(ERROR);                       // 退出程序，提示位置出错
     *e=L.Delem[i];
     return;
}
```

5）确定元素 e 是线性表的第几个元素(逻辑位置)：locateElem(L,e)

（1）静态分配代码如下。

```
unsigned  locateElem(SList L, eleType e) {
```

```
    for(int i=0;i<L.length-1;i++)
         if(L.Selem[i]==e)
        break;
      if(i!=L.length)
 return i+1;
         else
       return 0;
}
```

（2）动态分配代码如下。

```
unsigned  locateElem(DList L, eleType e) {
    for(int i=0;i<L.length-1;i++)
         if(L.Delem[i]==e)
         break;
    if(i!=L.length)
        return i+1;
        else
        return 0;
}
```

6）删除线性表的第 i 个元素，将其放在变量 e 中：listDelete(*L,i,*e)

（1）静态分配代码如下。

```
void listDelete(SList *L, unsigned i, eleType *e){
   if(L->length==0)          // 空表不能删除
      exit(EMPTY);           // 退出程序，提示空表
   if(i<1 || i>L->length)    // 非法逻辑位置
      exit(ERROR);           // 退出程序，提示位置出错
    *e=L->Selem[i-1]);       // 线性表的第 i 个元素存放于变量 e
    for(eleType *p=L->Selem[i]);p>=L->Selem[L->length-1];p++)
                             // 从第 i 个数据元素开始，依次将后面的元素前移一个位置
       *(p-1)=*p;
   --L->length;              // 表长减 1
    return;
}
```

（2）动态分配代码如下。

```
void listDelete(DList *L, unsigned i, eleType *e){
   if(L->length==0)          // 空表不能删除
      exit(EMPTY);           // 退出程序，提示空表
   if(i<1 || i>L->length)    // 非法逻辑位置
      exit(ERROR);           // 退出程序，提示位置出错
    *e=L->Delem[i-1]);       // 线性表的第 i 个元素存放于变量 e
    for(eleType *p=L->Delem[i]);p>=L->Delem[L->length-1];p++)
                             // 从第 i 个数据元素开始，依次将后面的元素前移一个位置
       *(p-1)=*p;
   --L->length;              // 表长减 1
    return;
}
```

7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值

（1）静态分配代码如下。

```
void printList(SList L){
    for(int i=0;i<L.length-1;i++)
       printf(L.Selem[i]);
    return;
}
```

（2）动态分配代码如下。

```
void printList(DList L){
    for(int i=0;i<L.length-1;i++)
       printf(L.Delem[i]);
    return;
}
```

8）线性表判空：emptyList (L)，为空返回非零值，非空返回 0

（1）静态分配代码如下。

```
int emptyList (SList L){
    return(!L.length);
}
```

（2）动态分配代码如下。

```
int emptyList (DList L){
    return(!L.length);
}
```

9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间

（1）静态分配代码如下。

```
void destroyList(SList *L){
     if(!L->Selem)       // 没有可销毁的内容
        exit(ERROR);
     free(L->Selem);
     L->length=0;
}
```

（2）动态分配代码如下。

```
void destroyList(DList *L){
     if(!L->Delem)       // 没有可销毁的内容
        exit(ERROR);
     free(L->Delem);
     L->length=0;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

5．应用场合

顺序表主要应用在数据量不大，且很少有插入、删除操作的场合。例如，学生基本信息。

2.3.2　链表

1．概念

链表是线性表的链式存储方式，线性表中每个数据元素单独申请和释放存储空间，逻辑相邻的元素其存储单元不必相邻，其特点如下。

（1）线性表的链式存储结构中，所有数据元素所占用的存储空间可以连续，也可以不连续。

（2）单个数据元素的存储单元不仅要存储数据元素的值（值域），还要存储数据元素之间的关系（指针域）。

（3）通过指针域反映逻辑关系。

（4）插入删除操作通过修改指针域完成，不用移动元素。

（5）查找操作需要遍历整个链表，无法实现随机查找。

（6）带头结点的链表，其头结点的值域可用来存放链表中数据元素个数（eleType 为数值型）、特殊值（例如最大值）等，也可不用。本书部分内容用其保存链表中数据元素的个数。

根据存储空间是整体申请还是插入操作时才申请，链表分为动态链表和静态链表。动态链表为进行插入操作时，为待插入元素申请所需的存储空间。静态链表为事先申请约定大小的内存空间，每个数据元素的指针域存放其前驱或后继的地址，所以逻辑上相邻的元素不一定放在相邻的位置上。

无论是动态链表，还是静态链表，都可构造单向链表或双向链表。

线性表的单向链表中每个数据元素一般包含一个指针域，用来存放该元素的直接后继的地址。线性表的双向链表中每个数据元素一般包含两个指针域，分别用来存放该元素的直接前驱和直接后继的地址。

无论单向链表，还是双向链表，均可通过首元素和末元素的指针域构造循环链表。无论单向链表，还是双向链表，均可包含头结点，即不存储任何数据元素，仅代表链表的结点。此时，空链表包含 1 个结点。可利用头结点存储链表的特有信息，例如元素个数、最大元素、最小元素等。

考生要重点掌握动态链表的非循环单向链表（单向链表）、单向循环链表、非循环双向链表（双向链表）、双向循环链表，以及静态链表的非循环静态链表（静态链表）。

2．单向链表

1）单向链表的存储结构

```
typedef struct Node{
     eleType      data;
     struct Node *next;
}LNode, *linkList;
linkList L; // 定义类型为 linkList 的指针变量 L，标识整个链表，指向“第一个结点”
```

不带头结点的单向链表如图 2.3 所示，带头结点的单向链表如图 2.4 所示。

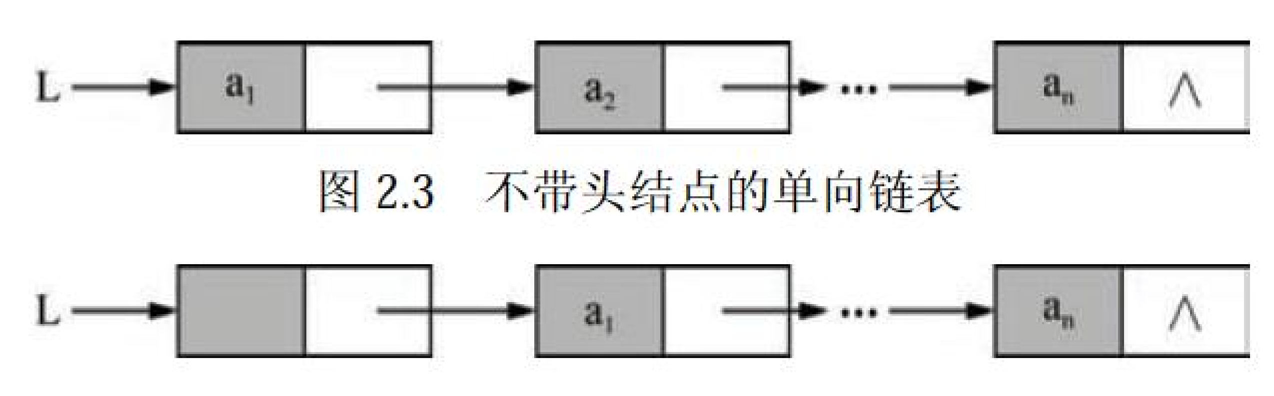


图 2.3　不带头结点的单向链表

图 2.4　带头结点的单向链表

2）单向链表的基本操作

（1）初始化线性表：initList1(*L)或 initList2(*L)。

① 不带头结点的代码如下。

```
void initList1(linkList L){
    L=NULL;                           // NULL 为空指针，可记为∧
    return;
}
```

② 带头结点的代码如下。

```
void initList2(linkList L){
    L=( LNode *)malloc(sizeof(LNode));
    if(!L)                            // 没有分配成功
        exit(OVERFLOW);               // 退出程序，提示溢出
    L->data=0;                        // 可以利用头结点的值域存放表长，不是必须
    L->next=NULL;                     // NULL 为空指针，可记为∧
    return;
}
```

（2）建立单链表：creatList**(L)，建立之前应该先初始化。

① 不带头结点的代码如下。

```
void creatList11(linkList L){     // 表头插入
    initList1(L);
    LNode *p;                         // p 为待插入链表的结点
    eleType x;                        // x 为待插入链表的结点的值
    scanf(&x);                        // 读取第一个元素的值
    while (x!=EOF){                   // EOF 为用户自定义的结束标记值
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=L;
        L=p;
        scanf(&x);                    // 读取下一个元素的值
    }
    return;
 }
```

或

```
 void creatList12(linkList L){    // 表尾插入
    initList1(L);
    LNode *p;                         // p 为待插入链表的结点
    LNode *tail;                      // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
```

```
    scanf(&x);                              // 读取第一个元素的值
    if (x==EOF)                             // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=( LNode *)malloc(sizeof(LNode));      // 建立含一个结点的链表
    p->data=x;
    p->next=NULL;
    L=p;
    tail=L;
    while (x!=EOF){                         // EOF 为用户自定义的结束标记值
        scanf(&x);                          // 读取下一个元素的值
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=NULL;                       // 也可用 p->next= tail->next
        tail->next=p;                       // p 链接至当前表尾 tail 之后
        tail=p;                             // p 为新的表尾 tail
    }
    return;
}
```

② 带头结点的代码如下。

```
void creatList21(linkList L){      // 表头插入
    initList2(L);
    LNode *p;                      // p 为待插入链表的结点
    eleType x;                     // x 为待插入链表的结点的值
    scanf(&x);                     // 读取第一个元素的值
    while (x!=EOF){                // EOF 为用户自定义的结束标记值
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=L->next;
        L->next =p;
        L->data++;                 // 不是必须，可以利用头结点的值域存放表长
        scanf(&x);                 // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList22(linkList L){      // 表尾插入
    initList2(L);
    LNode *p;                      // p 为待插入链表的结点
    LNode *tail=L;                 // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                     // 读取第一个元素的值
    while (x!=EOF){                // EOF 为用户自定义的结束标记值
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=NULL;              // 也可用 p->next= tail->next
        tail->next=p;              // p 链接至当前表尾 tail 之后
        tail=p;                    // p 为新的表尾 tail
        L->data++;                 // 可以利用头结点的值域存放表长，不是必须
        scanf(&x);                 // 读取下一个元素的值
    }
    return;
}
```

（3）查找元素：获取指定位置的结点或获取特定值的结点。可以按照位置查找，也可以按照值查找。

① 获取线性表的第 i 个元素：getElem1*(L,i)或 getElem2*(L,i,*e)。

● 不带头结点的代码如下。

```
LNode *getElem11 (linkList L,unsigned i){    // i 为逻辑位置
    LNode *p=L;                             // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
        return p;
 }
```

或

```
void getElem21(linkList L,unsigned i, LNode *e){    // i 为逻辑位置
    LNode *p=L;                             // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    e=p;
    return;
}
```

● 带头结点的代码如下。

```
LNode *getElem12 (linkList L,unsigned i){    // i 为逻辑位置
    LNode *p=L->next;                       // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    return p;
}
```

其中 while 循环部分也可改为

```
if(i<1 || i>L->data)
    return NULL
while(j!=i)
    p=p->next, j++;
```

或

```
void getElem22(linkList L,unsigned i, LNode *e){    // i 为逻辑位置
    LNode *p=L->next;                       // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
```

```
    }
    e=p;
    return;
}
```

其中 while 循环部分也可改为

```
if(i<1 || i>L->data)
    e=NULL;
while(j!=i)
    p=p->next, j++;
```

② 获取特定值的结点：getElem*(L,x)。

● 不带头结点的代码如下。

```
LNode *getElem3 (linkList L, eleType x){
    LNode *p=L;                              // p 为遍历链表元素结点的指针
    while(p!=NULL && p->data!=x)
        p=p->next;
    return p;
}
```

● 带头结点的代码如下。

```
LNode *getElem4 (linkList L,unsigned i){     // i 为逻辑位置
    LNode *p=L->next;                        // p 为遍历链表元素结点的指针
    while(p!=NULL && p->data!=x)
        p=p->next;
    return p;
}
```

（4）插入：在链表表头插入元素最方便，由于不支持随机访问，所以链表在指定位置插入元素意义不大，而且需要遍历链表，时间复杂度高。下面两个算法供考生比较。

① 在线性表的第 i 个元素（逻辑位置）插入一个值为 x 的元素：insertList1(L,i,x)。

```
void insertList1(linkList L, unsigned i, eleType x){  // 以不带头结点链表为例
        LNode *p,*q;                    // q 指向插入位置的前驱
        q=getElem11(L, i-1);            // 带头结点链表改为 q=getElem12(L,i-1)
        if(q==NULL)                     // i 值过大
            exit(ERROR);
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=q->next;
        q->next=p;
        // 带头结点链表可增加语句 L->data++来修正表长，不是必须
        return;
}
```

② 在线性表的表头插入一个值为 x 的元素：insertList2(L,x)。

```
void insertList2(linkList L, eleType x){         // 以不带头结点链表为例
        LNode *p;
        p=( LNode *)malloc(sizeof(LNode));
        p->data=x;
        p->next=L;                      // 带头结点链表改为 p->next=L->next
```

```
        L=p;                            // 带头结点链表改为 L->next=p
        // 带头结点链表可增加语句 L->data++来修正表长，不是必须
        return;
}
```

（5）求线性表的长度：listLength*(L)。

① 不带头结点的代码如下。

```
unsigned listLength1(linkList L){
    LNode *p=L;
    unsigned i=0;
    while(p!=NULL){
        i++;
        p=p->next;
    }
    return i;
}
```

② 带头结点的代码如下。

```
unsigned listLength1(linkList L){
    LNode *p=L->next;
    unsigned i=0;
    while(p!=NULL){
        i++;
        p=p->next;
    }
    return i;
}
```

（6）删除结点：删除线性表指定位置的结点或删除特定值的结点。

① 删除线性表的第 i 个结点：listDelete1(*L,i)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void listDelete1(linkList L,unsigned i){    // i 为逻辑位置
    LNode *p,*q;
    p=getElem11(L,i-1);                 // p 为待删结点的前驱结点
                                        // 带头结点链表改为 p=getElem21(L,i-1)
    if(p==NULL || p->next==NULL)        // 不存在第 i 个结点
        exit(ERROR);
    q=p->next;                          // q 为待删结点
    p->next=q->next;                    // q 脱离链表
    free(q);                            // 释放 q 所占存储空间
    return;
}
```

② 删除线性表中值为 x 的结点：listDelete2(*L,x)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

● 表中重复元素值的代码如下。

```
void listDelete2(linkList L, eleType x){
    LNode *p,*q;
    p=L;                  // 带头结点链表改为 p=L->next，p 最终指向待删结点的前驱结点
```

```
    if(p->data==x){   // 第一个结点的值为 x
        L=L->next;    // 带头结点链表改为 L->next=L->next->next
        free(p);
                      // 带头结点链表增加 L->data--，不是必须
        return;
    }
    q=p->next;
    while(q!=NULL && q->data!=x) // 定位待删结点及其前驱
        p=q,q=p->next;
    if(q==NULL)                  // 不存在值为 x 的结点
        exit(ERROR);
    p->next=q->next;             // q 脱离链表
    free(q);                     // 释放 q 所占存储空间
                                 // 带头结点链表增加 L->data--，不是必须
    return;
}
```

● 表中有重复元素值，删除所有目标元素的代码如下。

```
void listDelete2(linkList L, eleType x){
    LNode *p,*q;
    p=L;              // 带头结点链表改为 p=L->next，p 最终指向待删结点的前驱结点
    if(p->data==x){   // 第一个结点的值为 x
        L=L->next;    // 带头结点链表改为 L->next =L->next ->next
        free(p);
                            // 带头结点链表增加语句 L->data--，不是必须
        return;
    }
    q=p->next;
    while(q!=NULL){         // 定位待删结点及其前驱
        while(q!=NULL && q->data!=x){
            p=q;
            q=p->next;
        }
        if(q==NULL)         // 不存在值为 x 的结点
            exit(ERROR);
        else{               // 找到 1 个值为 x 的结点
            p->next=q->next;  // q 脱离链表
            free(q);        // 释放 q 所占存储空间
                            // 带头结点链表增加语句 L->data--，不是必须
        }
        q=p->next;          // 继续查找到下一个值为 x 的结点
    }
    return;
}
```

（7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void printList(linkList L){
    LNode *p;
    p=L;                    // 带头结点链表改为 p=L->next，p 指向待访问结点
    while(p!=NULL){
        printf(p->data);
        p=p->next;
```

```
    }
    return;
}
```

（8）线性表判空：emptyList (L)，为空返回非零值，非空返回 0，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
int emptyList(linkList L){
    if(L==NULL)           // 带头结点链表改为 if(L->next==NULL)
        return 1;
    return 0;
}
```

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，带头结点和不带头结点链表算法类似，下面代码以不带头结点链表为例。

```
void destroyList(linkList L){
    LNode *p,*q;
    p=L;                  // 带头结点链表改为 p=L->next，p 指向待删结点
    while(p!=NULL){
        q=p->next;
        free(p);
        p=q;
    }
    return;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

3）单向链表的应用场合

单向链表不必占用一块连续的内存空间。在查找、取值等静态操作比较少，增加、删除操作比较多的情况下，单向链表应用较多。

3. 单向循环链表

单向循环链表的最后一个结点的后继为第一个结点。

1）单向循环链表的存储结构

```
 Typedef struct RNode{
      eleType data;
      struct RNode  *next;
 }RLNode, *RlinkList;
RlinkList RLtail;
```

定义类型为 RlinkList 的指针变量 RLtail，标识整个链表，指向“最后一个结点”，其后继为头结点。对于循环链表来讲，定义尾结点既能表示最后一个结点，又能标识头结点，实现基本操作的算法比较方便。

不带头结点的单向链表如图 2.5 所示，带头结点的单向链表如图 2.6 所示。

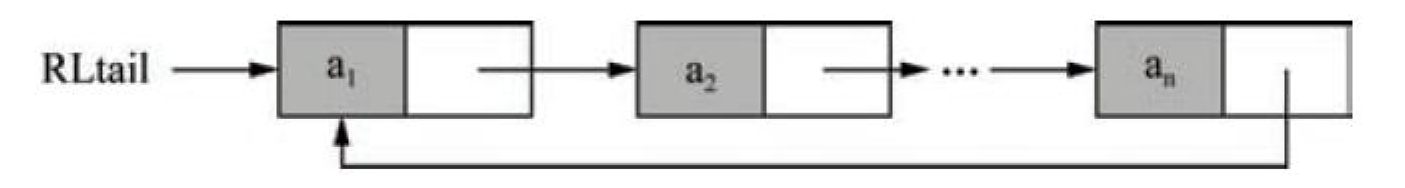


图 2.5　不带头结点的单向循环链表

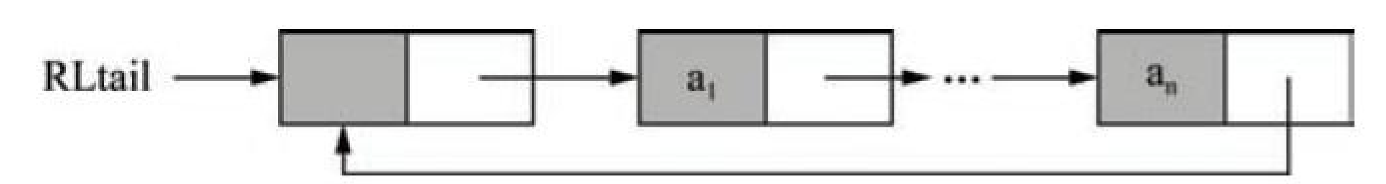


图 2.6　带头结点的单向循环链表

2）单向循环链表的基本操作

（1）初始化线性表：initList1(*L)或 initList2(*L)。

① 不带头结点的代码如下。

```
void initList1(RlinkList RLtail){
    RLtail=NULL;                        // NULL 为空指针，可记为∧
    return;
}
```

② 带头结点的代码如下。

```
void initList2(RlinkList RLtail){
    RLtail=(RLNode *)malloc(sizeof(RLNode));
    if(!RLtail)                         // 没有分配成功
        exit(OVERFLOW);                 // 退出程序，提示溢出
    RLtail->next=RLtail;
    RLtail->data=0;                     // 可以利用头结点的值域存放表长，不是必须
    return;
}
```

（2）建立单向循环链表：creatList**(L)，建立之前应该先初始化。

① 不带头结点的代码如下。

```
void creatList11(RlinkList RLtail){ // 表头插入
    initList1(RLtail);
    RLNode *p;                          // p 为待插入链表的结点
    eleType x;                          // x 为待插入链表的结点的值
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    RLtail=(RLNode *)malloc(sizeof(RLNode));
    RLtail->data=x;
    RLtail->next=RLtail;                // RLtail->next 为表头结点
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
        p=(RLNode *)malloc(sizeof(RLNode));
        p->data=x;
        p->next=RLtail->next;           // RLtail->next 为表头结点
        RLtail->next =p;
        scanf(&x);                      // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList12(RlinkList RLtail){ // 表尾插入
    initList1(RLtail);
    RLNode *p;                          // p 为待插入链表的结点
```

```
    eleType x;
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=(RLNode *)malloc(sizeof(RLNode));  // 建立含一个结点的链表
    p->data=x;
    p->next=p;
    RLtail=p;
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
        p=(RLNode *)malloc(sizeof(RLNode));
        p->data=x;
        p->next=RLtail->next;           // p 指向表头 RLtail->next，为新的表尾
        RLtail->next=p;                 // p 链接至当前表尾 RLtail 之后
        RLtail=p;                       // p 为新的表尾 RLtail
        scanf(&x);                      // 读取下一个元素的值
    }
    return;
}
```

② 带头结点的代码如下。

```
void creatList21(RlinkList RLtail){ // 表头插入
    initList2(RLtail);
    RLNode *p;                          // p 为待插入链表的结点
    eleType x;                          // x 为待插入链表的结点的值
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=(RLNode *)malloc(sizeof(RLNode));
    p->data=x;
    p->next=RLtail->next;
    RLtail=p;                           // RLtail 指向第一个插入的结点，即表尾
    RLtail->next->data++;               // 可利用头结点的值域存放表长，不是必须
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
        p=(RLNode *)malloc(sizeof(RLNode));
        p->data=x;
        p->next=RLtail->next->next;
        RLtail->next->next =p;
        RLtail->next->data++;           // 不是必须，可利用头结点的值域存放表长
        scanf(&x);                      // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList22(RlinkList RL){       // 表尾插入
    initList2(RL);
    RLNode *p;                          // p 为待插入链表的结点
    RLNode *tail=RL;                    // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
```

```
        exit(EMPTY);
    p=(RLNode *)malloc(sizeof(RLNode));
    p->data=x;
    p->next=RLtail->next;
    RLtail=p;                         // RLtail 指向第一个插入的结点，即表尾
    RLtail->next->data++;             // 可利用头结点的值域存放表长，不是必须
    scanf(&x);                        // 读取第二个元素的值
    while (x!=EOF){                   // EOF 为用户自定义的结束标记值
        p=(RLNode *)malloc(sizeof(RLNode));
        p->data=x;
        p->next=RLtail->next;         // p 指向表头
        RLtail->next=p;               // p 链接至当前表尾 RLtail 之后
        RLtail=p;                     // p 为新的表尾 RLtail
        L->data++;                    // 可以利用头结点的值域存放表长，不是必须
        scanf(&x);                    // 读取下一个元素的值
    }
    return;
}
```

（3）查找元素：获取指定位置的结点或获取特定值的结点。可以按照位置查找，也可以按照值查找。

① 获取线性表的第 i 个元素：getElem1*(L,i)或 getElem2*(L,i,*e)。

- 不带头结点的代码如下。

```
RLNode *getElem11(RlinkList RLtail,unsigned i){           // i 为逻辑位置
    RLNode *p=RLtail->next;                           // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=RLtail->next){
        p=p->next;
        j++;
    }
    if(i==j)
        return p;
    else
        return NULL;
}
```

或

```
void getElem21(RlinkList RLtail,unsigned i, RLNode *e){  // i 为逻辑位置
    RLNode *p=RLtail->next;                           // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(j!=i && p->next!=RLtail->next){
        p=p->next;
        j++;
    }
    if(i==j)
        e=p;
    else
        e=NULL;
    return;
}
```

● 带头结点的代码如下。

```
RLNode *getElem12(RlinkList RLtail,unsigned i){      // i 为逻辑位置
   RLNode *p=RLtail->next->next;             // p 为遍历链表元素结点的指针
   unsigned j=1;
   while(j!=i && p->next!=RLtail->next) {
      p=p->next;
      j++;
   }
   if(i==j)
      return p;
   else
      return NULL;
}
```

查找部分的代码也可改为

```
if(i<1 || i> RLtail->next ->data)
   return NULL;
while(j!=i)
   p=p->next, j++;
return p;
```

或

```
RLNode *getElem22(RlinkList RLtail,unsigned i, RLNode *e) // i 为逻辑位置
{
   RLNode *p=RLtail->next->next;         // p 为遍历链表元素结点的指针
   unsigned j=1;
   while(j!=i && p->next!=RLtail->next){
      p=p->next;
      j++;
   }
   if(i==j)
      e=p;
   else
      e=NULL;
   return;
}
```

查找部分的代码也可改为

```
if(i<1 || i> RLtail->next ->data)
   e=NULL;
while(j!=i)
   p=p->next, j++;
   e=p;
return;
```

② 获取特定值的结点：getElem*(L,x)，下面代码以不带头结点的循环链表为例，带头结点的算法类似。

```
RLNode *getElem3 (RlinkList RLtail, eleType x){
   RLNode *p=RLtail->next;
   // p 为遍历链表元素结点的指针，带头结点链表改为 RLNode *p=RLtail->next->next
   while(p->data!=x && p->next!=RLtail->next)
```

```
        p=p->next;
    if(p->data==x)
        return p;
    else
        return NULL;
}
```

（4）插入：在链表表头插入元素最方便，由于不支持随机访问，所以链表在指定位置插入元素意义不大，而且需要遍历链表，时间复杂度高。下面两个算法供考生比较。

① 在线性表的第 i 个元素(逻辑位置)插入一个值为 x 的元素：insertList1(L,i,x)。

```
void insertList1(RlinkList RLtail, unsigned i, eleType x){
               // 以不带头结点链表为例
    RLNode *p,*q;
    q=getElem11(RLtail, i);
               // 带头结点链表改为 q=getElem12(RLtail->next,i-1);
    if(q==NULL)
        exit(ERROR);
    p=(RLNode *)malloc(sizeofR(LNode));
    p->data=x;
    p->next=q->next;
    q->next=p;
               // 带头结点链表可增加语句RLtail->next->data++来修正表长，不是必须
    return;
}
```

② 在线性表的表头插入一个值为 x 的元素：insertList2(L,x)。

```
void insertList2(RlinkList RLtail, eleType x){ // 以不带头结点链表为例
    RLNode *p;
    p=(RLNode *)malloc(sizeof(RLNode));
    p->data=x;
    p->next=RLtail->next;
    // 带头结点链表改为 p->next=RLtail->next ->next
    RLtail->next=p;       // 带头结点链表改为 RLtail->next->next=p
    // 带头结点链表可增加语句RLtail->next->data++来修正表长，不是必须
    return;
}
```

（5）求线性表的长度：listLength*(L)。

① 不带头结点的代码如下。

```
unsigned listLength1(RlinkList RLtail){
    RLNode *p;
    unsigned i=1;
    if(RLtail==NULL)      // 空表
        return 0;
    p= RLtail->next;
    while(p->next!= RLtail->next){
        i++;
        p=p->next;
    }
    return i;
}
```

② 带头结点的代码如下。

```
unsigned listLength1(RlinkList RLtail){
    RLNode *p=RLtail->next;
    unsigned i=1;
    if(p->next==p)    // 空表
        return 0;
    while(p->next!=RLtail->next){
        i++;
        p=p->next;
    }
    return i;
}
```

（6）删除结点：删除线性表指定位置的结点或删除特定值的结点。

① 删除线性表的第 i 个结点：listDelete1(*L,i)，带头结点和不带头结点链表算法类似，以不带头结点为例。

```
void listDelete1(RlinkList RLtail,unsigned i){  // i 为逻辑位置
    RLNode *p,*q;
    p=getElem11(RLtail,i-1);                    // p 为待删结点的前驱结点
    // 带头结点链表改为 p=getElem21(RLtail,i-1)
    q= getElem11(RLtail,i);                     // q 为待删结点
    // 带头结点链表改为 q=getElem21(RLtail,i)
    if(p==NULL || q==NULL)                      // 不存在第 i 个结点
        exit(ERROR);
    p->next=q->next;                            // q 脱离链表
    free(q);                                    // 释放 q 所占存储空间
    return;
}
```

② 删除线性表中值为 x 的结点：listDelete2(*L,x)，带头结点和不带头结点链表算法类似，以不带头结点为例。

● 表中无重复元素值的情况代码如下：

```
void listDelete2(RlinkList RLtail, eleType x){
    RLNode *p,*q;
    p= RLtail->next;       // p 初值指向首元素结点，最终指向待删结点的前驱
                           // 带头结点链表改为 p= RLtail->next ->next
    if(p->data==x){        // 第一个结点的值为 x
        RLtail->next=RLtail->next->next;        // 删除首元素结点
                           // 带头结点链表改为
                           // RLtail->next->next=RLtail->next->next->next;
        free(p);
                           // 带头结点链表增加语句 RLtail->next->data--，不是必须
        return;
    }
    q=p->next;
    while(q!= RLtail->next && q->data!=x)       // 定位待删结点及其前驱
        p=q,q=p->next;
    if(q== RLtail->next)                        // 不存在值为 x 的结点
        exit(ERROR);
    p->next=q->next;                            // q 脱离链表
```

```
    free(q);                // 释放 q 所占存储空间
                            // 带头结点链表增加语句 RLtail->next->data--,不是必须
    return;
}
```

● 表中有重复元素，删除表中所有目标元素的代码如下：

```
void listDelete2(RlinkList RLtail, eleType x){
    RLNode *p,*q;
    p= RLtail->next;        // p 初值指向首元素结点，最终指向待删结点的前驱
                            // 带头结点链表改为 p=RLtail->next->next
    if(p->data==x){         // 第一个结点的值为 x
        RLtail->next=RLtail->next->next; // 删除首元素结点
                            // 带头结点链表改为
                            // RLtail->next->next=RLtail->next->next->next
        free(p);
                            // 带头结点链表增加语句 RLtail->next->data--，不是必须
        return;
    }
                            // 第一个结点的值不是 x
    q=p->next;
    while(q!=RLtail->next){   // 定位待删结点及其前驱
        while(q!=RLtail->next && q->data!=x){
            p=q;
            q=p->next;
        }
        if(q==RLtail->next)   // 不存在值为 x 的结点
            exit(ERROR);
        else{                 // 找到一个值为 x 的结点
            p->next=q->next;  // q 脱离链表
            free(q);          // 释放 q 所占存储空间
                              // 带头结点链表可增加语句 RLtail->next->data--,
                              // 不是必须
        }
        q=p->next;            // 继续查找到下一个值为 x 的结点
    }
    return;
}
```

（7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值，带头结点和不带头结点链表算法类似，以不带头结点为例。

```
void printList(RlinkList RLtail){
    RLNode *p;
    if(RLtail==NULL)          // 空表则直接退出
                              // 带头结点链表改为 if(RLtail->next == RLtail)
        exit(ERROR);
    p= RLtail->next;          // p 指向待访问结点，初值为首结点
                              // 带头结点链表改为 p= RLtail->next ->next
    do{
        printf(p->data);
        p=p->next;
    } while(p!= RLtail->next)
    return;
}
```

（8）线性表判空：emptyList (L)，为空返回非零值，非空返回 0，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
int emptyList(RlinkList RLtail){
    if(RLtail==NULL)        // 空表，带头结点链表改为 if(RLtail->next == RLtail)
        return 1;
    return 0;
}
```

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void destroyList(RlinkList RLtail){
    RLNode *p,*q;
    p=RLtail->next;         // p 指向待删结点
                            // 带头结点链表改为 p=RLtail->next->next
    while(p!=NULL){         // 带头结点链表改为 while(p!=p->next)
        q=p->next;
        free(p);
        p=q;
    }
    return;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

4．双向链表

1）双向链表的存储结构

```
typedef  struct  DNode{
     eleType       data;
     struct  DNode  *prior;      // 指向前驱
     struct  DNode  *next;       // 指向后继
}DLNode, *DlinkList;
DlinkList DL;
```

定义类型为 DlinkList 的指针变量 DL，标识整个链表，指向“第一个结点”。

不带头结点的双向链表如图 2.7 所示，带头结点的双向链表如图 2.8 所示。

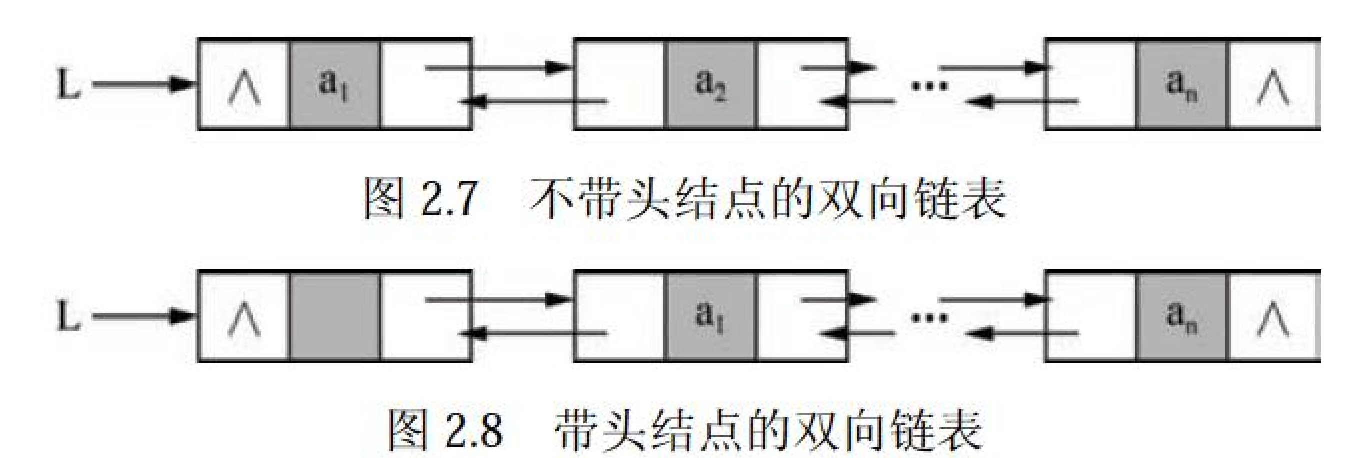


图 2.7　不带头结点的双向链表

图 2.8　带头结点的双向链表

2）双向链表的基本操作

（1）初始化线性表：initList1(*L)或 initList2(*L)。

① 不带头结点的代码如下。

```
void initList1(DlinkList DL){
    DL=NULL;                    // NULL 为空指针，可记为∧
```

```
    return;
}
```

② 带头结点的代码如下。

```
void initList2(DlinkList DL){
    DL=(DLNode *)malloc(sizeof(DLNode));
    if(!DL)                         // 没有分配成功
        exit(OVERFLOW);             // 退出程序，提示溢出
    DL->data=0;                     // 可以利用头结点的值域存放表长，不是必须
    DL->prior=NULL;                 // NULL 为空指针，可记为∧
    DL->next=NULL;
    return;
}
```

（2）建立双链表：creatList**(L)，建立之前应该先初始化。

① 不带头结点的代码如下。

```
void creatList11(DlinkList DL){ // 表头插入
    initList1(DL);
    DLNode *p;                      // p 为待插入链表的结点
    eleType x;                      // x 为待插入链表的结点的值
    scanf(&x);                      // 读取第一个元素的值
    while (x!=EOF){                 // EOF 为用户自定义的结束标记值
        p=(DLNode *)malloc(sizeof(DLNode));
        p->data=x;
        p->prior=NULL;
        p->next=DL;
        if(DL!=NULL)
            DL->prior=p;
        DL=p;
        scanf(&x);                  // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList12(DlinkList DL){ // 表尾插入
    initList1(DL);
    DLNode *p;                      // p 为待插入链表的结点
    DLNode *tail;                   // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                      // 读取第一个元素的值
    if (x==EOF)                     // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=(DLNode *)malloc(sizeof(DLNode)); // 建立含一个结点的链表
    p->data=x;
    p->prior=NULL;
    p->next=NULL;
    DL=p;
    tail=DL;
    scanf(&x);                      // 读取第二个元素的值
    while (x!=EOF){                 // EOF 为用户自定义的结束标记值
        p=(DLNode *)malloc(sizeof(DLNode));
```

```
        p->data=x;
        p->prior=tail;
        p->next=NULL;              // 也可以用 p->next= tail->next 语句
        tail->next=p;              // p 链接至当前表尾 tail 之后
        tail=p;                    // p 为新的表尾 tail
        scanf(&x);                 // 读取第一个元素的值
    }
    return;
}
```

② 带头结点的代码如下。

```
void creatList21(DlinkList DL){ // 表头插入
    initList2(DL);
    DLNode *p;                 // p 为待插入链表的结点
    eleType x;                 // x 为待插入链表的结点的值
    scanf(&x);                 // 读取第一个元素的值
    p=(DLNode *)malloc(sizeof(DLNode));
    p->data=x;
    p->prior=DL;
    p->next=NULL;
    DL->next=p;
    DL->data++;                // 可以利用头结点的值域存放表长，不是必须
    scanf(&x);                 // 读取第二个元素的值
    while (x!=EOF){            // EOF 为用户自定义的结束标记值
        p=(DLNode *)malloc(sizeof(DLNode));
        p->data=x;
        p->prior=DL;
        p->next=DL->next;
        DL->next->prior=p;
        DL->next=p;
        DL->data++;            // 可以利用头结点的值域存放表长，不是必须
        scanf(&x);             // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList22(DlinkList DL){ // 表尾插入
    initList2(DL);
    DLNode *p;                 // p 为待插入链表的结点
    DLNode *tail=DL;           // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                 // 读取第一个元素的值
    while (x!=EOF){            // EOF 为用户自定义的结束标记值
        p=(DLNode *)malloc(sizeof(DLNode));
        p->data=x;
        p->prior=tail;
        p->next=NULL;          // 也可使用 p->next= tail->next 语句
        tail->next=p;          // p 链接至当前表尾 tail 之后
        tail=p;                // p 为新的表尾 tail
        DL->data++;            // 可以利用头结点的值域存放表长，不是必须
        scanf(&x);             // 读取下一个元素的值
    }
```

```
    return;
}
```

（3）查找元素：获取指定位置的结点或获取特定值的结点。可以按照位置查找，也可以按照值查找。

① 获取线性表的第 i 个元素：getElem1*(L,i)或 getElem2*(L,i,*e)。

- 不带头结点的代码如下。

```
DLNode *getElem11 (DlinkList DL,unsigned i){ // i 为逻辑位置
    DLNode *p=DL;                             // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    return p;
}
```

或

```
void getElem21(DlinkList DL,unsigned i, DLNode *e){ // i 为逻辑位置
    DLNode *p=DL;                                    // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    e=p;
    return;
}
```

- 带头结点的代码如下。

```
DLNode *getElem12 (DlinkList DL,unsigned i){ // i 为逻辑位置
    DLNode *p=DL->next;                       // p 为遍历链表元素结点的指针
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    return p;
}
```

while 循环体部分的代码也可改为

```
if(i<1 || i>DL->data)
    return NULL;
while(j!=i)
    p=p->next, j++;
```

或

```
void getElem22(DlinkList DL,unsigned i, DLNode *e)      // i 为逻辑位置
{
    DLNode *p=DL->next;                                  // p 为遍历链表元素结点的指针
```

```
    unsigned j=1;
    while(p!=NULL && j!=i){
        p=p->next;
        j++;
    }
    e=p;
    return;
}
```

while 循环体部分的代码也可改为

```
if(i<1 || i>DL->data)
    e=NULL;
while(j!=i)
    p=p->next, j++;
```

② 获取特定值 x 的结点：getElem*(L,x)。

- 不带头结点的代码如下。

```
DLNode *getElem3(DlinkList DL, eleType x){
    DLNode *p=DL;                              // p 为遍历链表元素结点的指针
    while(p!=NULL && p->data!=x)
        p=p->next;
    return p;
}
```

- 带头结点的代码如下。

```
DLNode *getElem4(DlinkList DL,unsigned i){   // i 为逻辑位置
    DLNode *p=DL->next;                        // p 为遍历链表元素结点的指针
    while(p!=NULL && p->data!=x)
        p=p->next;
    return p;
}
```

（4）插入：在链表表头插入元素最方便，由于不支持随机访问，所以链表在指定位置插入元素意义不大，而且需要遍历链表，时间复杂度高。下面两个算法供考生比较。

① 在线性表的第 i 个元素（逻辑位置）插入一个值为 x 的元素：insertList1(L,i,x)。

```
void insertList1(DlinkList L, unsigned i, eleType x){
                                   // 以不带头结点链表为例
    DLNode *p,*q;                  // q 指向插入位置的前驱
    q=getElem11(DL, i-1);          // 带头结点链表改为 q=getElem12(DL,i-1)
    if(q==NULL)
        exit(ERROR);
    p=(DLNode *)malloc(sizeof(DLNode));
    p->data=x;
    p->prior=q;
    p->next=q->next;
    q->next->prior=p;
    q->next=p;
    // 带头结点链表可增加语句 DL->data++来修正表长，不是必须
    return;
}
```

② 在线性表的表头插入一个值为 x 的元素：insertList2(L,x)。

```
void insertList2(DlinkList DL, eleType x){  // 以不带头结点链表为例
    DLNode *p;
    p=(DLNode *)malloc(sizeof(DLNode));
    p->data=x;
    p->prior=NULL;              // 带头结点链表改为 p->prior=DL
    p->next=DL;                 // 带头结点链表改为 p->next=DL->next
    DL->prior=p;                // 带头结点链表改为 DL>next->prior=p
    DL=p;                       // 带头结点链表改为 DL->next=p
    // 带头结点链表可增加语句 DL->data++来修正表长，不是必须
    return;
}
```

（5）求线性表的长度：listLength*(L)。

① 不带头结点的代码如下。

```
unsigned listLength1(DlinkList DL){
    DLNode *p=DL;
    unsigned i=0;
    while(p!=NULL){
        i++;
        p=p->next;
    }
    return i;
}
```

② 带头结点的代码如下。

```
unsigned listLength1(DlinkList DL){
    DLNode *p=DL->next;
    unsigned i=0;
    while(p!=NULL){
        i++;
        p=p->next;
    }
    return i;
}
```

（6）删除结点：删除线性表指定位置的结点或删除特定值的结点。

① 删除线性表的第 i 个结点：listDelete1(*L,i)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void listDelete1(DlinkList DL,unsigned i){       // i 为逻辑位置
    DLNode *p,*q;
    p=getElem11(DL,i-1);        // p 为待删结点的前驱结点
                                // 带头结点链表改为 p=getElem21(DL,i-1)
    if(p==NULL || p->next==NULL) // 不存在第 i 个结点
        exit(ERROR);
    q=p->next;                  // q 为待删结点
    p->next=q->next;            // q 脱离链表
    q->next->prior=p
    free(q);                    // 释放 q 所占存储空间
                                // 带头结点链表增加语句 DL->data--，不是必须
```

```
    return;
}
```

② 删除线性表中值为 x 的结点：listDelete2(*L,x)，带头结点和不带头结点链表算法类似，以不带头结点为例。

● 表中无重复元素值的情况代码如下。

```
void listDelete2(DlinkList DL, eleType x){
    DLNode *p,*q;
    p=DL;                          // 带头结点链表改为p=DL->next，p最终指向待
                                   // 删结点的前驱结点
    if(p->data==x){                // 第一个结点的值为x
        DL=DL->next;               // 带头结点链表改为DL->next=DL->next->next
        DL->prior=NULL;
        free(p);                   // 带头结点链表增加语句DL->data--，不是必须
        return;
    }
    q=p->next;
    while(q!=NULL && q->data!=x) // 定位待删结点及其前驱
        p=q,q=p->next;
    if(q==NULL)                    // 不存在值为x的结点
        exit(ERROR);
    p->next=q->next;               // q脱离链表
    q->next->prior=p
    free(q);                       // 释放q所占存储空间
                                   // 带头结点链表增加语句DL->data--，不是必须
    return;
}
```

● 表中有重复元素值，删除所有目标元素的代码如下。

```
void listDelete2(DlinkList DL, eleType x){
    DLNode *p,*q;
    p=DL;                          // 带头结点链表改为p=DL->next，p最终指向待
                                   // 删结点的前驱结点
    if(p->data==x){                // 第一个结点的值为x
        DL=DL->next;               // 带头结点链表改为DL->next=DL->next->next,
        DL->prior=NULL;
        free(p);
                                   // 带头结点链表增加语句DL->data--，不是必须
    }
    q=p->next;
    while(q!=NULL){                // 定位待删结点及其前驱
        while(q!=NULL && q->data!=x){
            p=q;
            q=p->next;
        }
        if(q==NULL)                // 不存在值为x的结点
            exit(ERROR);
        else{                      // 找到一个值为x的结点
            p->next=q->next;       // q脱离链表
            q->next->prior=p
            free(q);               // 释放q所占存储空间
                                   // 带头结点链表增加语句L->data--，不是必须
```

```
        }
        q=p->next;              // 继续查找到下一个值为 x 的结点
    }
    return;
}
```

（7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值，带头结点和不带头结点的链表算法类似，下面代码以不带头结点的链表为例。

```
void printList(DlinkList DL){
    DLNode *p;
    p=DL;                       // 带头结点的链表改为 p=DL->next，p 指向待访问结点
    while(p!=NULL){
        printf(p->data);
        p=p->next;
    }
    return;
}
```

（8）线性表判空：emptyList (L)，为空返回非零值，非空返回 0，带头结点和不带头结点的链表算法类似，下面代码以不带头结点的链表为例。

```
int emptyList(DlinkList DL){
    if(DL==NULL)                // 带头结点的链表改为 if(DL->next==NULL)
        return 1;
    return 0;
}
```

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，带头结点和不带头结点的链表算法类似，下面代码以不带头结点的链表为例。

```
void destroyList(DlinkList DL){
    DLNode *p,*q;
    p=DL;                       // 带头结点的链表改为 p=DL->next，p 指向待删结点
    while(p!=NULL){
        q=p->next;
        free(p);
        p=q;
    }
    return;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

3）应用场合

相对单向链表，双向链表求前驱比较方便。

5. 双向循环链表

双向循环链表的最后一个结点的后继为第一个结点，第一个结点的前驱为最后一个结点。

1）双向循环链表的存储结构

```
typedef  struct  DRNode{
    eleType data;
```

```
    struct  DRNode  *prior;
    struct  DRNode  *next;
}DRLNode, *DRlinkList;
DRlinkList DRLtail;
```

定义类型为 DRlinkList 的指针变量 DRLtail，标识整个链表，指向“最后一个结点”，其后继为头结点，头结点的后继为 DRLtail。对于循环链表来讲，定义尾结点既能表示最后一个结点，又能标识头结点，实现基本操作的算法比较简单。

不带头结点的双向循环链表如图 2.9 所示，带头结点的双向循环链表如图 2.10 所示。



图 2.9　不带头结点的双向循环链表

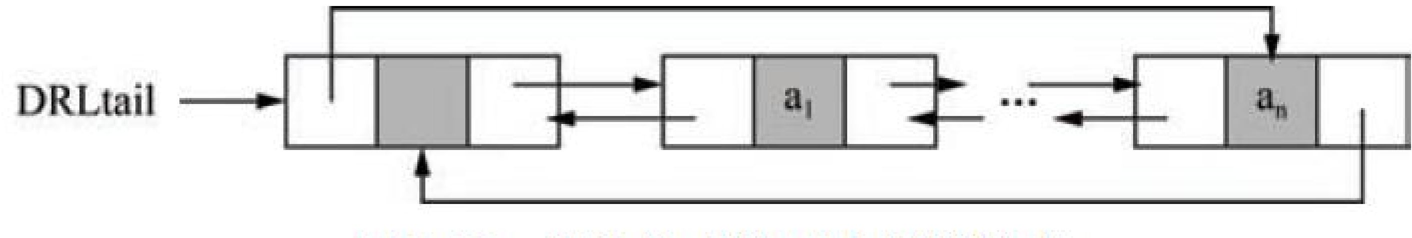


图 2.10　带头结点的双向循环链表

2）双向循环链表的基本操作

（1）初始化线性表：initList1(*L)或 initList2(*L)。

① 不带头结点的代码如下。

```
void initList1(DRlinkList DRLtail){
    DRLtail=NULL;                    // NULL 为空指针，可记为∧
    return;
}
```

② 带头结点的代码如下。

```
void initList2(DRlinkList DRLtail){
    DRLtail=(DRLNode *)malloc(sizeof(DRLNode));
    if(!DRLtail)                     // 没有分配成功
        exit(OVERFLOW);              // 退出程序，提示溢出
    RLtail->prior=RLtail;
    RLtail->next=RLtail;
    RLtail->data=0;                  // 可以利用头结点的值域存放表长，不是必须
    return;
}
```

（2）建立双向循环链表：creatList**(L)，建立之前应该先初始化。

① 不带头结点的代码如下。

```
void creatList11(DRlinkList DRLtail){    // 表头插入
    initList1(DRLtail);
    DRLNode *p;                      // p 为待插入链表的结点
    eleType x;                       // x 为待插入链表的结点的值
    scanf(&x);                       // 读取第一个元素的值
    if (x==EOF)                      // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
```

```
    DRLtail=(DRLNode *)malloc(sizeof(DRLNode));
    RL-tail>data=x;
    RLtail->prior=RLtail;
    RLtail->next=RLtail;                // RLtail->next 为表头结点
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
        p=(DRLNode *)malloc(sizeof(DRLNode));
        p->data=x;
        p->prior=DRLtail
        p->next=DRLtail->next;          // DRLtail->next 为表头结点
        DRLtail->next->prior=p;
        DRLtail->next=p;
        scanf(&x);                      // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList12(DRlinkList DRLtail){   // 表尾插入
    initList1(DRLtail);
    DRLNode *p;                         // p 为待插入链表的结点
    eleType x;
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=(DRLNode *)malloc(sizeof(DRLNode)); // 建立含一个结点的链表
    p->data=x;
    p->prior=p;
    p->next=p;
    DRLtail=p;
    scanf(&x);                          // 读取第二个元素的值
    while (x!=EOF){                     // EOF 为用户自定义的结束标记值
        p=(DRLNode *)malloc(sizeof(DRLNode));
        p->data=x;
        p->prior=DRLtail;
        p->next=DRLtail->next;      // p 指向表头 DRLtail->next，为新的表尾
        DRLtail->next=p;            // p 链接至当前表尾 DRLtail 之后
        RLtail->next->prior=p;      // p 链接至当前表头 DRLtail->next 之前
        DRLtail=p;                  // p 为新的表尾 DRLtail
        scanf(&x);                  // 读取下一个元素的值
    }
    return;
}
```

② 带头结点的代码如下。

```
void creatList21(DRlinkList DRLtail){   // 表头插入
    initList2(DRLtail);
    DRLNode *p;                         // p 为待插入链表的结点
    eleType x;                          // x 为待插入链表的结点的值
    scanf(&x);                          // 读取第一个元素的值
    if (x==EOF)                         // 如果第一个元素的值为结束标记，退出
        exit(EMPTY);
    p=(DRLNode *)malloc(sizeof(DRLNode));
```

```
    p->data=x;
    p->prior=DRLtail;
    p->next=DRLtail;
    DRLtail->prior=p;
    DRLtail->next=p;
    DRLtail=p;                        // DRLtail 指向第一个插入的结点，即表尾
    DRLtail->next->data++;            // 可利用头结点的值域存放表长，不是必须
    scanf(&x);                        // 读取第二个元素的值
    while (x!=EOF){                   // EOF 为用户自定义的结束标记值
       p=(DRLNode *)malloc(sizeof(DRLNode));
       p->data=x;
       p->prior=DRLtail->next;        // p 的前驱为表头结点 DRLtail->next
       p->next=DRLtail->next->next;   // p 的后继为首元素结点 DRLtail->next
       DRLtail->next->next->prior=p;
       DRLtail->next->next =p;
       DRLtail->next->data++;         // 可利用头结点的值域存放表长，不是必须
       scanf(&x);                     // 读取下一个元素的值
    }
    return;
}
```

或

```
void creatList22(DRlinkList DRL){ // 表尾插入
    initList2(DRL);
    DRLNode *p;                   // p 为待插入链表的结点
    DRLNode *tail=DRL;            // tail 为链表的尾结点，初值为刚初始化的链表
    eleType x;
    scanf(&x);                    // 读取第一个元素的值
    if (x==EOF)                   // 如果第一个元素的值为结束标记，退出
       exit(EMPTY);
    p=(DRLNode *)malloc(sizeof(DRLNode));
    p->data=x;
    p->prior=DRLtail;
    p->next=DRLtail;
    DRLtail->prior =p;
    DRLtail->next=p;
    DRLtail=p;                    // DRLtail 指向第一个插入的结点，即表尾
    DRLtail->next->data++;        // 可利用头结点的值域存放表长，不是必须
    scanf(&x);                    // 读取第二个元素的值
    while (x!=EOF){               // EOF 为用户自定义的结束标记值
       p=(DRLNode *)malloc(sizeof(DRLNode));
       p->data=x;
       p->prior= DRLtail;
       p->next=DRLtail->next;     // p 指向表头
       DRLtail->next=p;           // p 链接至当前表尾 DRLtail 之后
       DRLtail->next->prior=p;    // p 为表头 DRLtail->next 的前驱
       DRLtail=p;                 // p 为新的表尾 DRLtail
       L->data++;                 // 可以利用头结点的值域存放表长，不是必须
       scanf(&x);                 // 读取下一个元素的值
    }
    return;
}
```

（3）查找元素：获取指定位置的结点或获取特定值的结点。可以按照位置查找，也可以按照值查找。

① 获取线性表的第 i 个元素：getElem1*(L,i)或 getElem2*(L,i,*e)。

● 不带头结点的代码如下。

```
DRLNode *getElem11(DRlinkList DRLtail,unsigned i){
                                                    // i 为逻辑位置
   DRLNode *p=DRLtail->next;                        // p 为遍历链表元素结点的指针
   unsigned j=1;
   while(j!=i && p->next!=DRLtail->next){
      p=p->next;
      j++;
   }
   if(i==j)
      return p;
   else
      return NULL;
}
```

或

```
void getElem21(DRlinkList DRLtail,unsigned i, DRLNode *e){
                                                    // i 为逻辑位置
   DRLNode *p=DRLtail->next;                        //  p 为遍历链表元素结点的指针
   unsigned j=1;
   while(j!=i && p->next!=RLtail->next){
      p=p->next;
        j++;
   }
   if(i==j)
      e=p;
   else
      e=NULL;
   return;
}
```

● 带头结点的实现可参考前面算法，利用头结点值域存放的元素个数修改程序。

```
DRLNode *getElem12(DRlinkList DRLtail,unsigned i){
                                                    // i 为逻辑位置
   DRLNode *p=DRLtail->next->next;                  // p 为遍历链表元素结点的指针
   unsigned j=1;
   while(j!=i && p->next!=DRLtail->next){
      p=p->next;
      j++;
   }
   if(i==j)
      return p;
   else
      return NULL;
}
```

或

```
DRLNode *getElem22(DRlinkList DRLtail,unsigned i, DRLNode *e){
```

```
                                                     // i 为逻辑位置
   DRLNode *p=DRLtail->next->next;                   // p 为遍历链表元素结点的指针
   unsigned j=1;
   while(j!=i && p->next!=DRLtail->next){
      p=p->next;
      j++;
   }
   if(i==j)
      e=p;
   else
      e=NULL;
   return;
}
```

② 获取特定值的结点：getElem*(L,x)，下面代码以不带头结点循环链表为例，带头结点的算法类似。

```
DRLNode *getElem3 (DRlinkList DRLtail, eleType x){
   DRLNode *p=DRLtail->next;     // p 为遍历链表元素结点的指针，带头结点链表
                                 // 改为 dRLNode *p=DRLtail->next->next
   while(p->data!=x && p->next!=DRLtail->next)
      p=p->next;
   if(p->data==x)
      return p;
   else
      return NULL;
}
```

（4）插入：在链表表头插入元素最方便，由于不支持随机访问，所以链表在指定位置插入元素意义不大，而且需要遍历链表，时间复杂度高。下面两个算法供考生比较。

① 在线性表的第 i 个元素（逻辑位置）插入一个值为 x 的元素：insertList1(L,i,x)。

```
void insertList1(DRlinkList DRLtail, unsigned i, eleType x){
                                 // 以不带头结点链表为例
   DRLNode *p,*q;
   q=getElem11(DRLtail, i-1);    // 带头结点链表改为 q=getElem12(DRLtail,i-1)
   if(q==NULL)
      exit(ERROR);
   p=(DRLNode *)malloc(sizeof(RLNode));
   p->data=x;
   p->prior=q;
   p->next=q->next;
   q->next->prior=p;
   q->next=p;
   // 带头结点链表可增加语句 DRLtail->next->data++来修正表长，不是必须
   return;
}
```

② 在线性表的表头插入一个值为 x 的元素：insertList2(L,x)。

```
void insertList2(DRlinkList DRLtail, eleType x){ // 以不带头结点链表为例
   DRLNode *p;
   p=(DRLNode *)malloc(sizeof(DRLNode));
   p->data=x;
   p->prior=DRLtail;     // 带头结点链表改为 p->prior=DRLtail->next
```

```
    p->next=DRLtail->next;
                        // 带头结点链表改为 p->next=DRLtail->next ->next
    DRLtail->next->prior=p;
    DRLtail->next=p;    // 带头结点链表改为 DRLtail->next->next=p
            // 带头结点链表可增加语句 DRLtail->next->data++来修正表长，不是必须
    return;
}
```

（5）求线性表的长度：listLength*(L)。

① 不带头结点的代码如下。

```
unsigned listLength1(DRlinkList DRLtail){
    DRLNode *p;
    unsigned i=1;
    if(DRLtail==NULL)           // 空表
        return 0;
    p=DRLtail->next;
    while(p->next!=DRLtail->next){
        i++;
        p=p->next;
    }
    return i;
}
```

② 带头结点的代码如下。

```
unsigned listLength1(DRlinkList DRLtail){
    DRLNode *p=DRLtail->next;   // 可直接 return DRLtail->next->data
    unsigned i=1;
    if(p->next==p)              // 空表
        return 0;
    while(p->next!=DRLtail->next){
        i++;
        p=p->next;
    }
    return i;
}
```

（6）删除结点：删除线性表指定位置的结点或删除特定值的结点。

① 删除线性表的第 i 个结点：listDelete1(*L,i)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void listDelete1(DRlinkList DRLtail,unsigned i){    // i 为逻辑位置
    DRLNode *p,*q;
    p=getElem11(DRLtail,i-1);   // p 为待删结点的前驱结点
                                // 带头结点链表改为 p=getElem21(DRLtail,i-1)
    q= getElem11(DRLtail,i);    // q 为待删结点
                                // 带头结点链表改为 q=getElem21(DRLtail,i);
    if(p==NULL || q==NULL)      // 不存在第 i 个结点
        exit(ERROR);
    p->next=q->next;            // q 脱离链表
    q->next->prior=p;
                                // 带头结点链表可增加语句
                                // DRLtail->next->data--，不是必须
```

```
    free(q);                        // 释放 q 所占存储空间
    return;
}
```

② 删除线性表中值为 x 的结点：listDelete2(*L,x)，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

● 链表中无重复元素值的代码如下。

```
void listDelete2(DRlinkList DRLtail, eleType x){
    DRLNode *p,*q;
    p=DRLtail->next;                //p 开始指向首元素结点，最终指向待删结点的前驱
                                    // 带头结点链表改为 p=DRLtail->next->next
    if(p->data==x){                 // 第一个结点的值为 x
        DRLtail->next=p->next;      // 删除首元素结点
        p->next->prior=DRLtail;
                        // 带头结点链表改为 DRLtail->next->next=p->next
                        //  p->next->prior =DRLtail->next
        free(p);
                        // 带头结点链表可增加语句 DRLtail->next->data--，不是必须
        return;
    }
                                    // 第一个结点的值不是 x
    q=p->next;
    while(q!=DRLtail->next && q->data!=x)    // 定位待删结点及其前驱
        p=q,q=p->next;
    if(q==DRLtail->next)            // 不存在值为 x 的结点
        exit(ERROR);
    p->next=q->next;                // q 脱离链表
    q->next->prior=p;
    free(q);                        // 释放 q 所占存储空间
                        // 带头结点链表增加 DRLtail->next->data--; 不是必须
    return;
}
```

● 表中有重复元素值，删除所有目标元素的代码如下。

```
void listDelete2(DRlinkList DRLtail, eleType x){
    DRLNode *p,*q;
    p=DRLtail->next;                //p 开始指向首元素结点，最终指向待删结点的前驱
                                    // 带头结点链表改为 p=DRLtail->next->next
    if(p->data==x){                 // 第一个结点的值为 x
        DRLtail->next=p->next;      // 删除首元素结点
        p->next->prior=DRLtail;
                        // 带头结点链表改为 DRLtail->next->next=p->next;
                        // p->next->prior =DRLtail->next;
        free(p);
                        // 带头结点链表可增加语句 DRLtail->next->data--，不是必须
    }
    q=p->next;
    while(q!=DRLtail->next){        // 定位待删结点及其前驱
        while(q!=DRLtail->next && q->data!=x){
            p=q;
            q=p->next;
        }
```

```
        if(q==DRLtail->next)  // 不存在值为 x 的结点
            exit(ERROR);
        else{                 // 找到一个值为 x 的结点
            p->next=q->next;  // q 脱离链表
            q->next->prior=p;
            free(q);          // 释放 q 所占存储空间
                              // 带头结点链表可增加语句 DRLtail->next->data--,
                              // 不是必须
        }
        q=p->next;            // 继续查找到下一个值为 x 的结点
    }
    return;
}
```

（7）遍历线性表：printList(L,visit())，假设遍历为输出线性表的数据元素值，带头结点和不带头结点的链表算法类似，下面代码以不带头结点的链表为例。

```
void printList(DRlinkList DRLtail){
    DRLNode *p;
    if(DRLtail==NULL)         // 空表即退出，带头结点链表改为 if(DRLtail->next
                              // ==DRLtail)
        exit(ERROR);
    p=DRLtail->next;          // p 指向待访问结点，初值为首结点
                              // 带头结点链表改为 p=DRLtail->next->next
    do{
        printf(p->data);
        p=p->next;
    } while(p!=DRLtail->next)
    return;
}
```

（8）线性表判空：emptyList(L)，为空返回非零值，非空返回 0，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
int emptyList(DRlinkList DRLtail){
    if(DRLtail==NULL)  //空表。带头结点链表改为 if(DRLtail->next ==DRLtail)
        return 1;
    return 0;
}
```

（9）销毁线性表：destroyList(*L)，释放线性表 L 占用的空间，带头结点和不带头结点链表算法类似，下面代码以不带头结点的链表为例。

```
void destroyList(DRlinkList DRLtail){
    DRLNode *p,*q;
    p=DRLtail->next;          // p 指向待删结点
                              // 带头结点链表改为 p=DRLtail->next->next
    while(p!=NULL){           // 带头结点链表改为 while(p!=p->next)
        q=p->next;
        free(p);
        p=q;
    }
    return;
}
```

请考生自行分析上述基本操作的时间复杂度和空间复杂度。

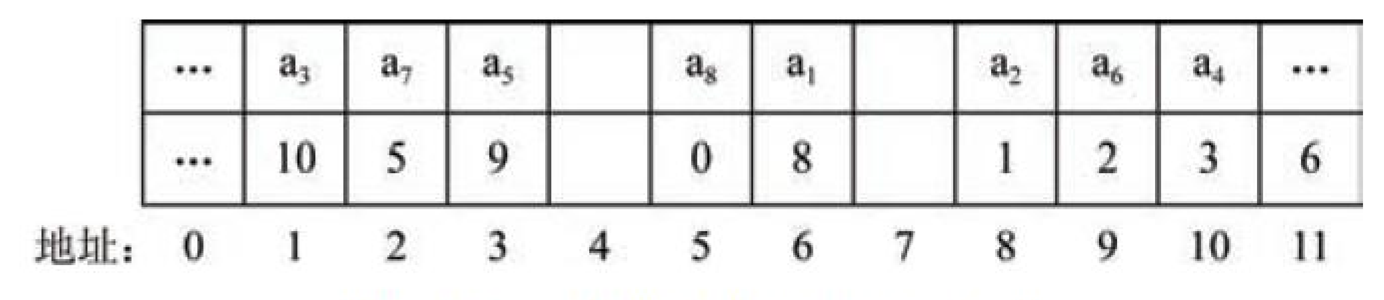
3）应用场合

从任何一个结点开始均可以很方便地操作其前驱和后继结点。

6. 静态链表

1）静态链表的存储结构

静态链表通过结构体数组存储，结构体包括两个成员，一个存放数据元素，另外一个存放该元素后继的地址，其存储示意图如图 2.11 所示，起始地址为 11，结束地址为 0。



地址：	0	1	2	3	4	5	6	7	8	9	10	11
	…	a_3	a_7	a_5		a_8	a_1		a_2	a_6	a_4	…
	…	10	5	9		0	8		1	2	3	6

图 2.11　静态链表存储结构示意图

存储结构描述如下。

```
#define maxSize 1000
typedef struct{
   eleType data;
   unsigned next;
}Node,SLink;
Slink SL[maxSize];
```

其中，SL[0]表示头结点，SL[0].data 可用来存放静态链表元素个数（eleType 为数值型）、特数值（例如最大值）等，也可不用，SL[0].next 指示第一个数据元素的存储位置。最后一个元素的 next 值为 0，表示静态链表结束。插入、删除操作通过修改元素的 next 值完成，无须移动数据元素。

2）静态链表的基本操作

（1）初始化：initSLink(*SL)，将 SL[0].next 设置为结束标记值 0，其他元素的 next 设置为-1，表示没有存放数据元素。

```
void initSLink(SLink *SL){
    SL[0].next=0;
                              // 如果 eleType 为数值型，也可加入语句 SL[0].data=0,
                              // 表示元素个数为 0，方便用户求表长
   for(unsigned i=1;i<maxSize)
      SL[i].next=-1;
   return;
}
```

（2）在线性表的第 i 个元素之前插入一个元素 x：insertSLink(*SL,i,x)，如果当前静态链表的存储空间没有用完，可进行插入操作，否则退出。算法流程如下。

- 遍历链表存储元素空间的所有 next 域，找到一个值为-1 的 SL[j].next，将 x 存入 SL[j].data。
- 通过 SL 的 next 域遍历静态链表，定位到第 i-1 个元素位置。
- 将 SL[j]插入到 SL[i-1]后面：SL[j].next=SL[i-1].next, SL[i-1].next=j。
- SL[0].data 如果存放元素个数的话，SL[0].data++。

```
void insertSLink(SLink *SL,unsigned i,eleType x){
    for(unsigned j=1;j<maxSize;j++)
        if(SL[j].next==-1)
            break;
    if(j==maxSize-1)
        exit(OVERFLOW);
    SL[j].data=x;
    unsigned h=0,k;                    // h 统计元素个数，k 表示元素的 next 值
    for(k=SL[0].next;h<i-1;k=SL[k].next)
        h++;
    SL[j].next=SL[k].next;             // k 为第 i-1 个元素的位置
    SL[k].next=j;
                                       // SL[0].data 如果存放元素个数的话，
                                       // SL[0].data++
    return;
}
```

（3）求线性表的长度 SListLength(SL)。

```
unsigned SListLength(SLink SL){        // unsigned 也可改为 int
    unsigned i=0,k;                    // i 统计元素个数，k 表示元素的 next 值
    for(k=SL[0].next;k!=0;k=SL[k].next)
        i++;
    return i;
                                       // SL[0].data 中如果存放元素个数的话，该
                                       // 算法仅需一条语句：return SL[0].data
}
```

（4）获取线性表的第 i 个元素：getElem1(SL,i)或 getElem2(SL,i,x)，i 为逻辑位置，取值范围 1～maxSize−1。

```
eleType getElem1(SLink SL, unsigned i){
    unsigned k,j=1;
    for(k=SL[0].next;j<i,k!=0;k=SL[k].next)
        j++;
    if(k==0)
        exit(ERROR);
    return SL[j].data;
}
```

或

```
void getElem2(SLink SL, unsigned i, eleType *x){
    unsigned k,j=1;
    for(k=SL[0].next;j<i,k!=0;k=SL[k].next)
        j++;
    if(k==0)
        exit(ERROR);
    *x=SL[j].data;
    return;
}
```

（5）确定 x 是线性表的第几个元素：locateElem(SL,x)。

```
unsigned locateElem(SLink *SL,eleType x){
    unsigned k,j=1;
    for(k=SL[0].next;k!=0,SL[k].data!=x;k=SL[k].next)
```

```
        j++;
    if(k==0)
        exit(ERROR);
    return j;
}
```

（6）删除线性表的第 i 个元素：SListDelete(*L,i,*x)。

```
void SLlistDelete(SLink *SL,unsigned i,eleType* x){
    unsigned k,j=1;
    for(k=SL[0].next;j<i-1,k!=0;k=SL[k].next)     // 定位第 i-1 个元素
        j++;
    if(k==0)
        exit(ERROR);
    *x=SL[k].next.data;       // 待删元素值存放于 x
    j=SL[k].next.next;        // 待删元素后继位置 SL[k].next.next 存放于 j
    SL[k].next.next=-1;       // 待删元素后继位置置-1，释放其存储空间
    SL[k].next=j;             // 第 i-1 个元素的后继修改为已删元素的后继
    return;
}
```

（7）遍历线性表：printSLink(SL)，假设遍历为输出元素的值。

```
void printSLink(SLink SL){
    for(unsigned k=SL[0].next;k!=0;k=SL[k].next)
        printf(SL[k].data);
    return;
}
```

（8）线性表判空：emptySLink(*SL)。

```
int emptySLink(SLink *SL){
    return !SL[0].next; // SL[0].next=0 为空表，返回真；否则返回假
}
```

3）应用场合

静态链表适用于没有指针类型的编程环境下需要使用链表的场合。

2.4 栈

2.4.1 定义

栈只能在一个固定端进行插入和删除操作的线性表，固定端称为栈顶，不能进行操作的一端称为栈底。

栈具有“后进先出”特性，即最后进入栈的元素最先出栈，是一种运算受限于一端的线性表。

当栈中没有任何元素时，称为栈空；栈的存储空间用完时，称为栈满。栈空栈满的

条件依赖不同存储结构。

入栈和出栈是栈的基本操作，通过修改栈顶指针完成，如图 2.12 所示。

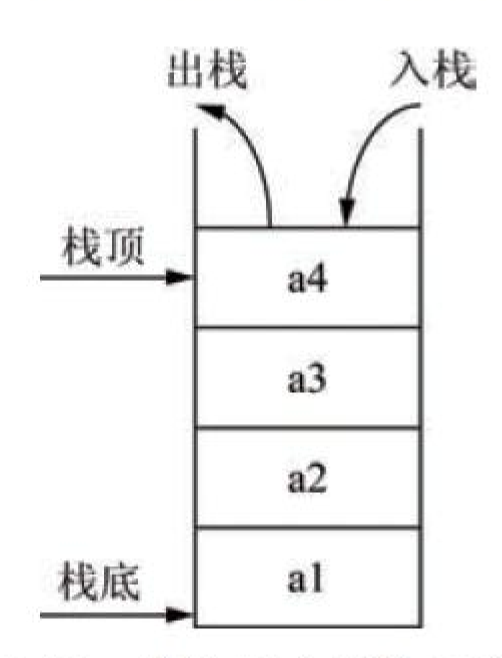


图 2.12　栈的基本操作示意图

栈的操作过程中需注意以下几点。

（1）栈底和栈顶为相对概念，当栈底的存储地址处于栈的小地址端，则入栈时栈顶指针增加，出栈时栈顶指针减小；反之，当栈底的存储地址处于栈的大地址端，则入栈时栈顶指针减小，出栈时栈顶指针增加。机器不同，约定不同。

（2）栈顶指针可能指向栈顶元素存放的位置，也可能指向新栈顶元素可以存放的位置。

（3）栈空、栈满的判定和存储结构的约定有关。

（4）数据元素的所有可能出栈顺序需要熟练掌握。

栈的抽象数据类型 ADT 描述如下。

```
ADT stack{
    数据对象 D：D={ ai| ai|∈eleSet, i=1,2,…,n, n≥0}
    数据关系 R：R={< ai-1, ai > | | ai-1, ai∈D, i=2,…,n, n≥0}
    约定 an 为栈顶，a1 为栈底。
    基本操作有如下几种。
        void initStack(*s)          // 初始化堆栈，构造空栈
        unsigned emptyStack(s)      // 判栈空，栈空返回 1，否则返回 0
        unsigned fullStack(s)       // 判栈满，栈满返回 1，否则返回 0
        void push(*s,x)             // 入栈，将 x 压入堆栈，形成新的栈顶
        void pop(*s,*x)             // 出栈，将栈顶元素出栈并存入 x，形成新栈顶
        void getTop(s,*x)           // 将栈顶元素存入 x，不出栈
} ADT stack
```

2.4.2　存储结构

1. 顺序存储结构

1）顺序存储结构的基本原理

利用地址连续的存储空间依次存放从栈底到栈顶的所有数据元素，称为顺序栈，可通过一维数组实现，见图 2.13，图中假设栈空间为 4 个存储单元。

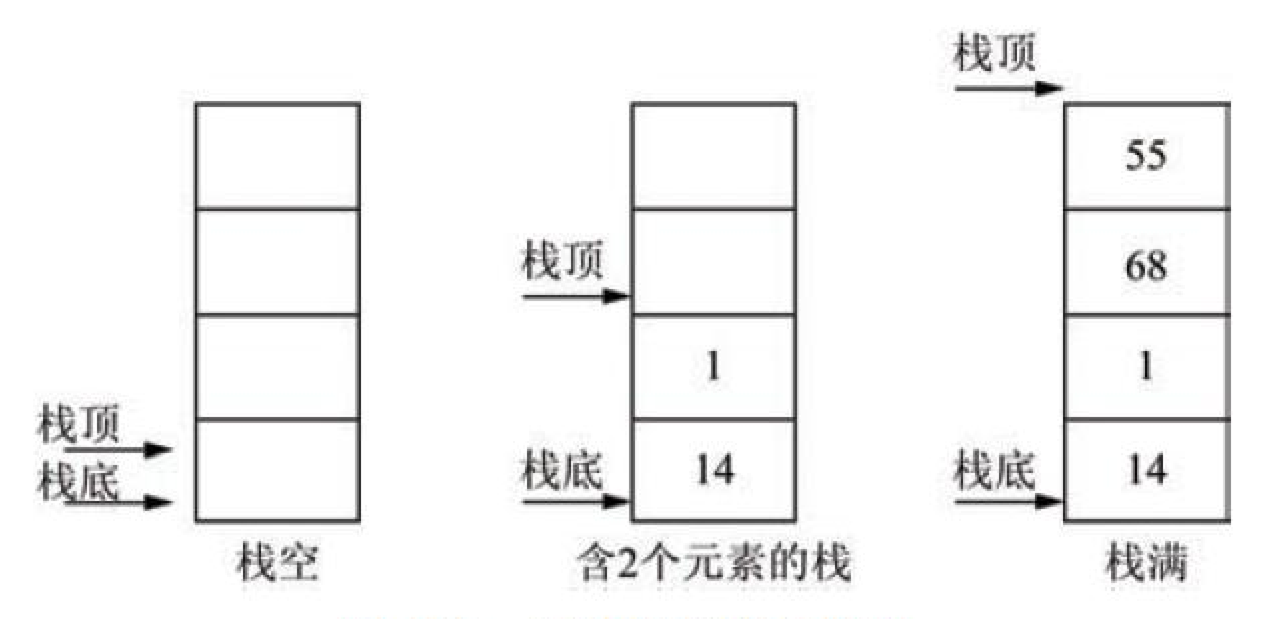


图 2.13　栈的顺序存储结构

顺序栈的描述如下。

```
#define maxStackSize 1000
typedef  struct
{
    eleType data[maxStackSize];      // 存放数据元素
    int top;                         // 指示栈顶位置
}SqStack;
```

2）顺序存储结构的基本操作

（1）初始化栈：initStack(*s)，约定入栈前先修改栈顶指针，所有栈顶指针初值为-1。

```
void initStack(SqStack *s){
   if(!(s=(SqStack *)malloc(sizeof(SqStack))))
      exit(ERROR);
   s->top=-1;
   return;
}
```

（2）判栈空：emptyStack(s)。

```
unsigned emptyStack(SqStack s){
   return s.top==-1?1:0;  // 当 s.top 为-1 时，返回 1，表示栈空为真；否则返回 0
}
```

（3）判栈满：fullStack(s)。

```
unsigned fullStack(SqStack s){
   return s.top== maxStackSize-1?1:0;
}
```

（4）入栈：push(*s,x)。

```
void push(SqStack*s,eleType x){
   if(fullStack(s))
      exit(OVERFLOW);
   s->top++;
   s[s->top]=x;
   return;
}
```

（5）出栈：pop(*s,*x)。

```
void pop(SqStack *s,eleType *x){
```

```
    if(emptyStack(s))
        exit(EMPTY);
    *x=s[s->top--];
    return;
}
```

（6）取栈顶元素：getTop(s,*x)。

```
void getTop (SqStack *s,eleType *x){
    if(emptyStack(s))
        exit(EMPTY);
    *x=s[s->top];
    return;
}
```

2．链式存储结构

1）链式存储结构的基本原理

如果栈中数据元素个数不确定，可以选用链式存储结构，元素入栈时为其申请存储空间，栈中各元素的存储单元地址不必连续，如图 2.14 所示。

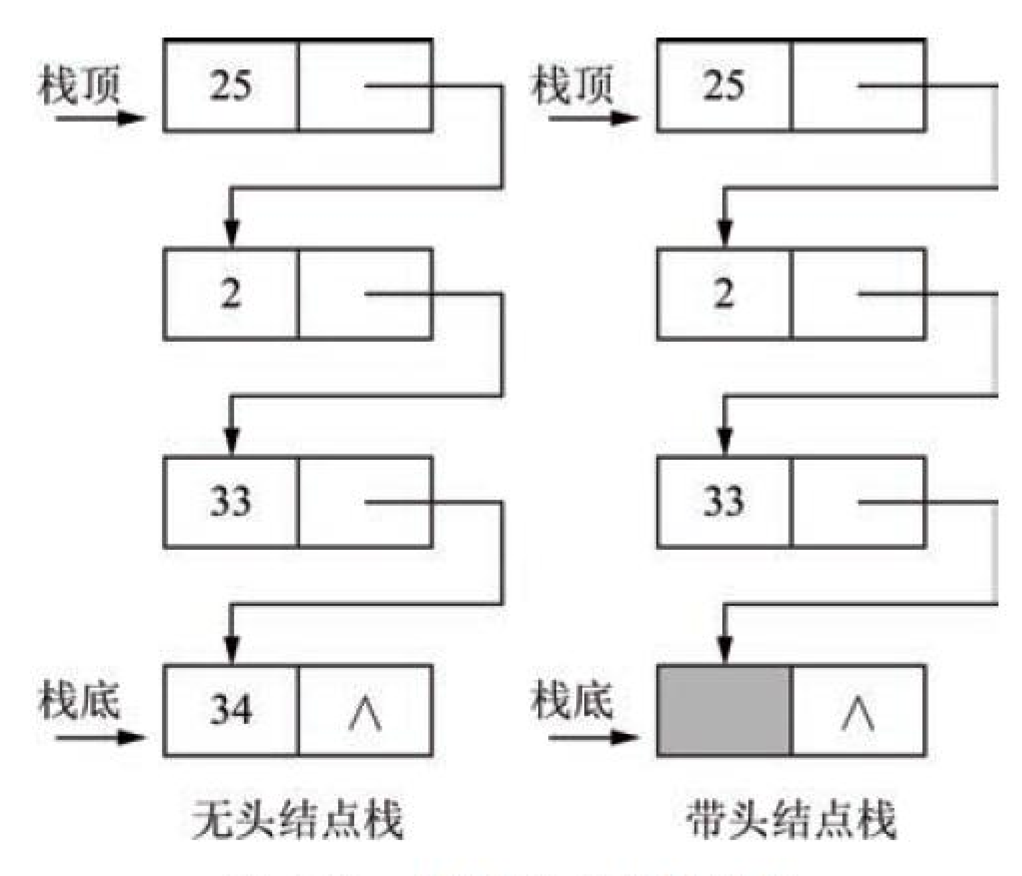


图 2.14　栈的链式存储结构

链栈的描述如下。

```
typedef  struct  Node
{
    eleType  data;              // 存放数据元素
    struct  Node  *next;        // 指示栈中当前元素后继的存放位置
}stackNode,*linkStack;
linkStack top;                  // top 为栈顶指针，为表头结点，方便栈的操作实现
```

2）链式存储结构的基本操作，以不带头结点栈为例

（1）初始化栈：initStack(*top)，约定入栈前先修改栈顶指针，所有栈顶指针初值为-1。

```
void initStack(linkStack top){
    top=NULL;
    // 带头结点栈为 if(!(top=(stackNode *)malloc(sizeof(stackNode))))
    // exit(ERROR)
    return;
}
```

（2）判栈空：emptyStack(top)。

```
unsigned emptyStack(linkStack top){
    return !top; // 当 top 为 NULL 时，返回 1，表示栈空为真；否则返回 0
}
```

（3）入栈：push(*top,x)。

```
void push(linkStack top,eleType x){
    stackNode *s;
    s=(stackNode *)malloc(sizeof(stackNode));
    s->data=x;
    s->next=top;
    top=s;
    return;
}
```

（4）出栈：pop(*top,*x)。

```
void pop(linkStack top,eleType *x){
    stackNode *s;
    if(emptyStack(top))
        exit(EMPTY);
    *x=top->data;
    s=top;
    top=top->next;
    free(s);
    return;
}
```

（5）取栈顶元素：getTop(top,*x)。

```
void getTop (linkStack top,eleType *x){
    if(emptyStack(top))
        exit(EMPTY);
    *x=top->data;
    return;
}
```

2.4.3 应用

栈在编程中的用途很广，下面讲述常用的两种情况。

1．符号匹配

程序中经常用到一些成对出现的符号，例如括号，对其进行是否成对的检验是基本的编译问题，可利用栈的特点进行检测。每扫描到一个左括号，将其入栈，扫描到右括号，将其出栈。扫描结束，若栈非空，返回 0，说明左右括号个数不一致，否则返回 1，说明二者个数匹配。

（1）顺序栈算法描述如下。

```
unsigned  signMatch(){
    SqStack s;
    char ch,t;
```

```
    initStack(&s);
    while((ch=getchar()!=EOF)){
        if(ch==' (')
            push(&s,ch);
        if(ch==' ) '){
            if(emptyStack(s))
                return 0;
            pop(&s,&t);    // 也可以用 pop(&s,&ch)，这样可以省略变量 t
        }
    }
    if(emptyStack(s))
        return 1;
}
```

（2）链栈算法描述如下。

```
unsigned signMatch(){
    linkStack top;
    char ch;
    initStack(top);
    while((ch=getchar()!=EOF)){
        if(ch==' (')
            push(top,ch);
        if(ch==') '){
            if(emptyStack(top))
                return 0;
            pop(top,&t);    // 也可以用 pop(top,&ch)，这样可以省略变量 t
        }
    }
    if(emptyStack(top))
        return 1;
}
```

2．整数进制转换

以十进制转为八进制为例，其他进制的转换类似。例如将一个十进制数字 n 转换为八进制数，用 n 除以 8 取余数，所有余数倒序输出即为对应的八进制数。

（1）顺序栈算法描述如下。

```
void ten2Eight(unsigned n){
    SqStack s;
    unsigned x;
    initStack(&s);
    while(n){
        push(&s,n%8);
        n/=8;
    }
    while(!emptyStack(s)){
        pop(&s,&x);
        printf("%u",x);
    }
    return;
}
```

（2）链栈算法描述如下。

```
void ten2Eight(unsigned n){
```

```
    linkStack top;
    unsigned x;
    initStack(top);
    while(n){
        push(top,n%8);
        n/=8;
    }
    while(!emptyStack(top)){
        pop(top,&x);
        printf("%u",x);
    }
    return;
}
```

2.5 队　　列

2.5.1 定义

队列是只能在一个固定端进行插入操作、另外一端进行删除操作的线性表，能够进行插入操作的端称为队尾，能够进行删除操作的端称为队头。

队列具有“先进先出”特性，即最先进入队列的元素也最先出队，是一种运算受限的线性表。

当队列中没有任何元素时，称为队空；队列的存储空间用完时，称为队满。队空队满的条件依赖不同存储结构。

入队和出队是队列的基本操作，分别通过修改队尾指针和队头指针完成，如图 2.15 所示。

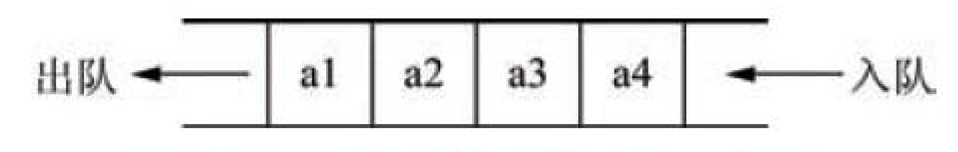


图 2.15　队列的基本操作示意图

队列的使用过程中要注意以下几点。

（1）队头指针可能指向队首元素存放的位置，也可能指向队首元素位置的前一个位置。

（2）队尾指针可能指向队尾元素存放的位置，也可能指向队尾元素位置的下一个位置。

（3）队空、队满的判定和存储结构的约定有关。

（4）和堆栈结合，数据元素的所有可能出栈、出队顺序需要熟练掌握。

队列的抽象数据类型 ADT 描述如下。

```
ADTquene{
    数据对象 D：D={ a_i| a_i|∈eleSet, i=1,2,…,n, n≥0}
    数据关系 R：R={< a_{i-1}, a_i > | | a_{i-1}, a_i∈D, i=2,…,n, n≥0}
    约定 a_1 为队头，a_n 为队尾。
    基本操作如下。
        void initQuene(*q)              // 初始化队列，构造空队
```

```
        unsigned emptyQuene(*q)        // 判断队是否空，队空返回 1，否则返回 0
        unsigned fullQuene(*q)         // 判断队是否满，队满返回 1，否则返回 0
        void enQuene(*q,x)             // 入队，将 x 存放到队尾后面，形成新的队尾
        void deQuene(*q,*x)            // 出队，将队首元素出队并存入 x，形成新的队首
        unsigned lengthQuene(*q)       // 返回队列元素的个数
} ADT queen
```

2.5.2　存储结构

1. 顺序存储结构

1）顺序存储结构的基本原理

利用地址连续的存储空间依次存放从队首到队尾的所有元素，称为顺序队列，可通过一维数组实现，见图 2.16，假设队列空间为 4 个存储单元。

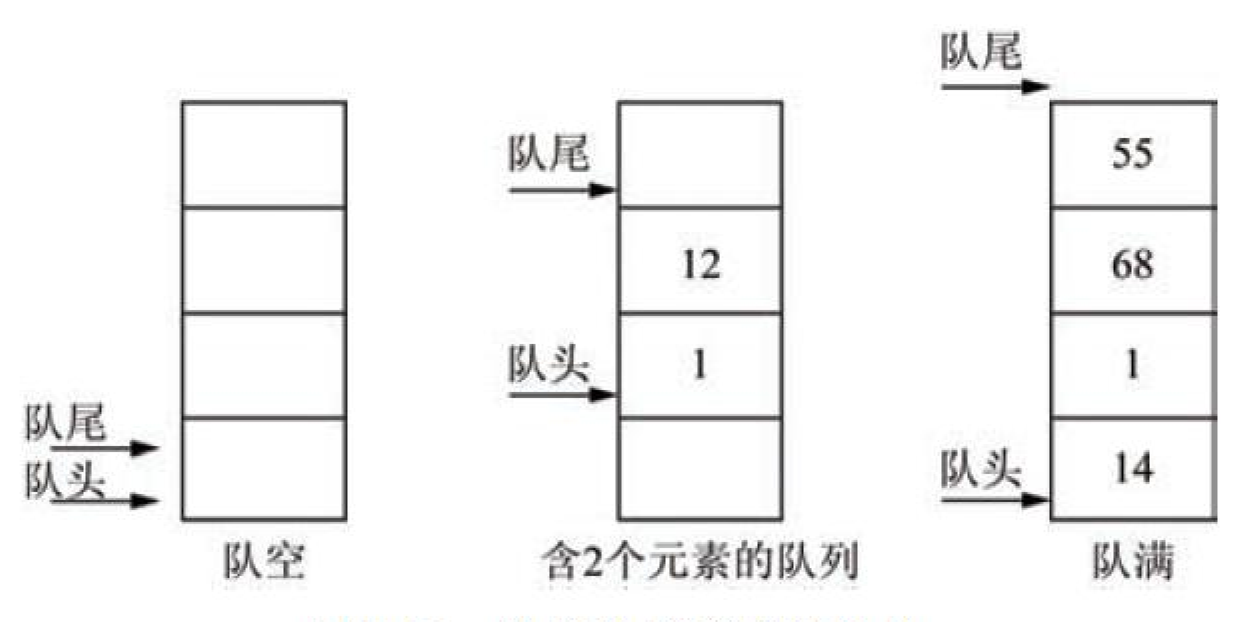


图 2.16　队列的顺序存储结构

顺序队列的描述如下。

```
#define maxQueneSize 1000
typedef struct
{
    eleType data[maxQueneSize];     // 存放数据元素
    int front;                      // 指示队头元素存放位置
    int rear;                       // 指示队尾元素存放位置的下一个位置
}SqQuene;
```

由于队列的基本操作入队和出队同方向修改队头指针和队尾指针，即同加或同减，随着入队、出队操作的随机进行，可能出现假溢出现象，即队尾已达最大值，无法入队，但队列中仍有空闲单元的情况。如图 2.17 所示，队列共占据 4 个存储单元，目前队列仅有一个元素 55，但是由于 rear 已经超出顺序队范围，无法进行入队操作。其中 base 为顺序队基址。

解决假溢出的办法是构建循环队列，如图 2.18 所示。入队时队尾指针的变化为

```
rear=(rear+1)% maxQueneSize;
```

出队时队头指针的变化为

```
front=(front+1)% maxQueneSize;
```

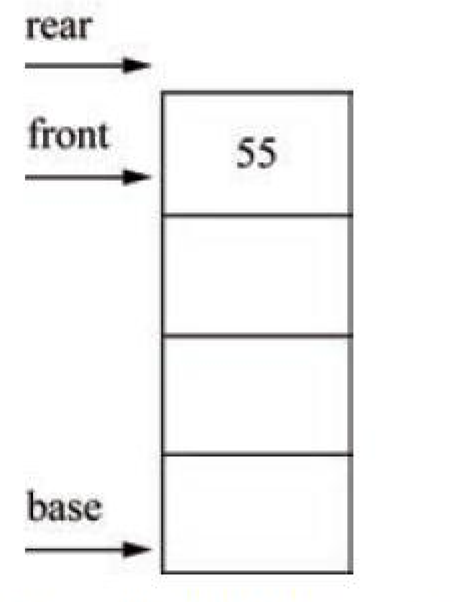



图 2.17　队列的假溢出现象

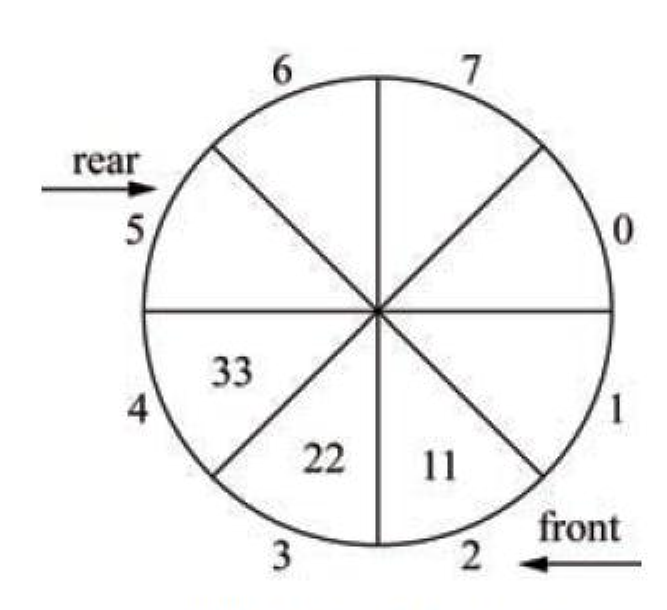


图 2.18　循环队列

2）循环队列的基本操作

（1）初始化循环队列：initQuene(*q)，约定入队前先修改队首指针，所以队首和队尾指针初值为 0。

```
void initQuene(SqQuene *q){
    if(!(q=(SqQuene *)malloc(sizeof(SqQuene))))
        exit(ERROR);
    q->front=q->rear=0;
    return;
}
```

（2）判循环队列空：emptyQuene(*q)。

```
unsigned emptyQuene(SqQuene q){
    if(q->rear==q->front)         // 循环队列为空的条件为 q->rear==q->front
        return 1;
    return 0;
}
```

（3）判循环队列满：fullQuene(*q)。

```
unsigned fullQuene(SqQuene *q){
    if((q->rear+1) % maxSize==q->front)
     // 循环队列满的条件为 (q->rear+1) % maxSize==q->front，即队尾追上队头
        return 1;
    return 0;
}
```

（4）入队：enQueue(*q,x)。

```
void enQueue(SqQueue *q,eleType x){
    if(fullQuene(q))
        exit(OVERFLOW);
    q->data[(q->rear) % maxQueueSize] =x;
    q->rear=(q->rear+1) % maxQueueSize;
    return;
}
```

（5）出队：deQueue(*q,*x)。

```
void deQueue(SqQueue *q,eleType *x){
    if(emptyQuene(q))
        exit(EMPTY);
    *x=q->data[q->front];
```

```
    q->front=(q-> front+1) % maxQueueSize;
    return;
}
```

（6）求队列长度：lengthQuene(*q)。

```
unsigned deQueue(SqQueue *q){
    return (q->rear-q->front+maxQueueSize) % maxQueueSize;
}
```

2. 链式存储结构

1）链式存储结构的基本原理

如果队列中数据元素个数不确定，可以选用链式存储结构，元素入队时为其申请存储空间，出队时释放其所占存储单元，队中各元素的存储单元地址不必连续，如图 2.19 所示。

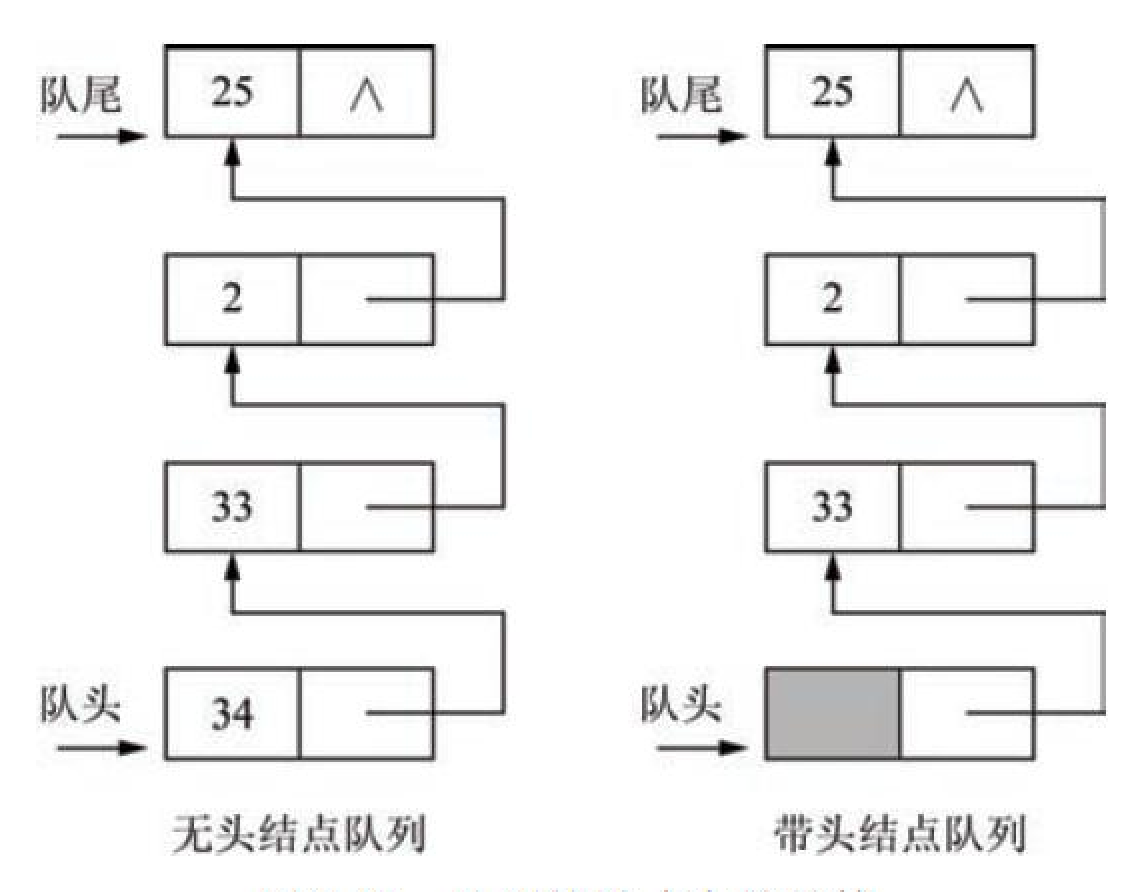


图 2.19　队列的链式存储结构

队列的链式存储结构描述如下。

```
typedef struct qNode{
    eleType data;              // 存放数据元素
    struct qNode *next;        // 指示队列中当前元素后继的存放位置
}queneNode;
typedef struct {
    queneNode *front;          // 存放队头指针
    queneNode *rear;           // 存放队尾指针
}linkQuene;
```

2）链式存储结构的基本操作

（1）初始化队列：initQuene(*q)，创建带头结点的空队。

```
void initQuene(linkQuene *q){
    queneNode *s;
    if(!q=(linkQuene *)malloc(sizeof(linkQuene)))
        exit(ERROR);
    if(!s=(queneNode *)malloc(sizeof(queneNode)))
        exit(ERROR);
    s->next=NULL;
```

```
    q->front=q->rear=s;
    return;
}
```

（2）判队空：emptyQuene(*q)。

```
unsigned emptyQuene(linkQuene *q){
    return q->front==q->rear;
}
```

（3）入队：enQuene(*q,x)。

```
void enQuene ( linkQuene *q, eleType  x){
    queneNode *s;
    if(!s=(queneNode *)malloc(sizeof(queneNode)))
        exit(ERROR);
    s->data=x;
    s->next=NULL;
    q->rear->next=s;                // s 链至队尾之后
    q->rear=s;                      // s 为新的队尾
    return;
}
```

（4）出队：deQuene(*q,*x)。

```
void deQuene ( linkQuene  *q, eleType  *x) {
    queneNode *s= q->front->next;
    if( emptyQuene (q) )
        exit(EMPTY);
    *x=s->data;
    q->front->next = s->next;
    free(s);
    if (q->front->next == NULL)     // 队空
        q->rear= q->front;
    return;
}
```

2.5.3 应用

下面以键盘输入来讲述队列的应用

键盘缓冲区用来暂存未处理的输入码，为“循环队列”结构。缓冲区满时，按键无效。

假设某程序包含两个进程，其中一个进程在屏幕上连续显示字符“Process A”，同时程序不断检测键盘是否有输入。如果有，则读入用户输入的字符并存入输入缓冲区。在用户输入过程中，输入的字符并不立即显示在屏幕上。输入“$”表示第一个进程结束。

另外一个进程从缓冲区中读取输入的字符并显示在屏幕上。

第二个进程结束后，程序又进入第一个进程，重新显示字符“Process A”，同时用户可以继续输入字符，直到输入“#”键，结束第一个进程，同时整个程序结束。

```
void simKeyBuffer(){                // 模拟键盘的循环输入缓冲区
    char ch1,ch2;
```

```
    SqQueue  q;
    initQueue (&q);                    // 初始化队列
    for( ; ; ){
       for( ; ; ){                     // 第一个进程代码开始
          printf("Process A\n");
          if (hitKeyboard() ){         // 有按键
             ch1=readChar();           // 读入一个字符
             if ( fullQuene(q) ) {
                printf("输入缓冲区已满，第一个进程被迫中止。\n");
                break;                 // 第一个进程非正常中止
             }
             enterQueue (&q,ch1);
          }
          if(ch1=='$' || ch1=='#')
              break;                   // 第一个进程正常结束
       }                               // 第一个进程代码结束
       while (!emptyQuene(q)) {        // 第二个进程代码开始
          deQueue (&q,&ch2);
          putchar(ch2);                // 按照输入顺序，输出输入缓冲区中的字符
       }
       if(ch1=='#')
            break;                     // 整个程序结束
       else
          ch1=' ';                     // ch1 赋值为非结束标记（$ 或 #），程序继续
    }
 }
```

2.6 特殊矩阵

矩阵是在科学工程计算中常见的数学模型之一，m 行 n 列排列有 m×n 个类型相同的数据元素。由于计算机的存储空间是线性的，所以一般程序设计语言通过一维数组存放二维的矩阵，有行优先（存完第 i 行的 n 个数据元素再存储第 i+1 行的 n 个数据元素）和列优先（存完第 i 列的 m 个数据元素再存储第 i+1 列的 m 个数据元素）两种存储方式。

矩阵通过二维数组 a[m][n]以行优先方式存放，其数组元素 a[i][j] 的地址计算公式如下：

```
a[i][j].addr= a[0][0].addr+(i*n+j)*sizeof(eleType) i∈[0,m-1]，j∈[0,n-1]
```

a[i][j]在一维存储空间的相对位置为第 i*n+j 个结点。

列优先方式存放时，数组元素 a[i][j] 的地址计算公式如下：

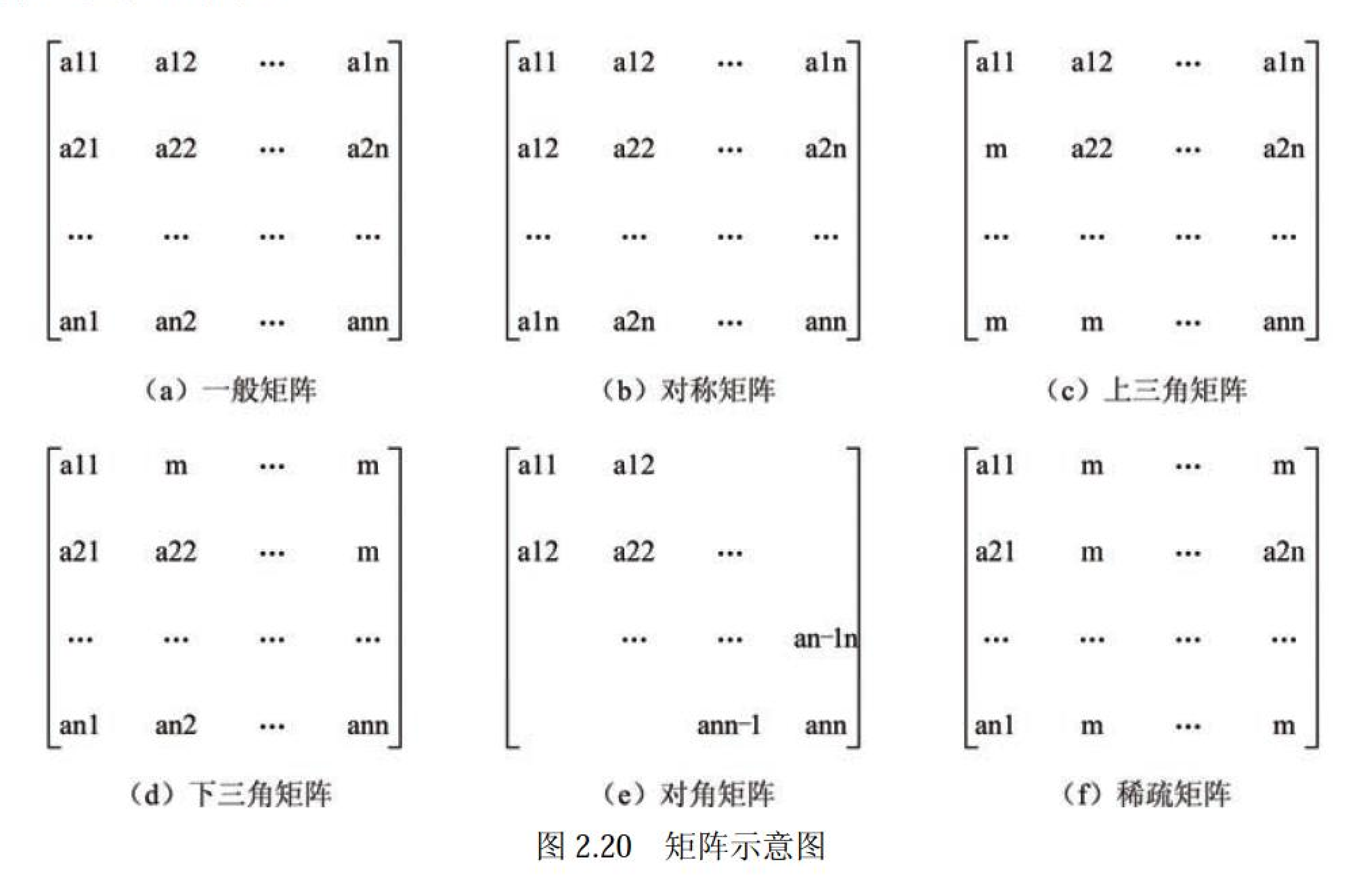
```
a[i][j].addr= a[0][0].addr+(j*m+i)*sizeof(eleType) i∈[0,m-1]，j∈[0,n-1]
```

a[i][j]在一维存储空间的相对位置为第 j*m+i 个结点。

当 m 和 n 较大时，需要占用大量的连续存储空间。但是实际应用中存在一些特殊的

矩阵，其中的数据元素值的分布有规律，或尽管矩阵包含数据元素非常多，但是其中不同值的元素（例如非 0 元素）很少。特殊矩阵存储时可为多个值相同的元素分配一个存储空间，对零元不分配空间，以降低矩阵对存储容量的需求。本节详细讲述特殊矩阵的存储方式及应用。

图 2.20 为示例矩阵，i 和 j 表示逻辑位置，i∈[1,m]，j∈[1,n]，m=n，图中 aij 对应的数组元素为 a[i-1][j-1]。其中，图 2.20（a）为一般矩阵；图 2.20（b）为对称矩阵，数据元素的值沿对角线对称出现；图 2.20（c）为上三角矩阵，对角线及其上方的元素值没有规律，但对角线以下所有数据元素具有相同的值；图 2.20（d）为下三角矩阵，对角线及其下方的元素值没有规律，但对角线以上所有数据元素具有相同的值；图 2.20（e）为对角矩阵，对角线及其同列最相邻的上下行两个元素值没有规律，其余数据元素具有相同的值；图 2.20（f）为稀疏矩阵，不同值的数据元素分布没有规律，但数量很少。



（a）一般矩阵　（b）对称矩阵　（c）上三角矩阵

（d）下三角矩阵　（e）对角矩阵　（f）稀疏矩阵

图 2.20　矩阵示意图

2.6.1　对称矩阵

n 阶对称矩阵，其数据元素满足 $a_{ij}=a_{ji}$，i, j∈[1,n]，存储于二维数组 a[n][n]，其对应数组元素为 a[k][h]，其中 k=i-1，h=j-1，(k,h∈[0,n-1])，压缩存储时对称的两个相同的值元素只存储一个，则第 i 行仅存储 i 个元素（假设存储下三角的数据元素），可以极大降低矩阵对存储空间的需求。n*n 个元素需要的存储结点（数组元素）数为：

$$1+2+\cdots+n=n(n+1)/2$$

图 2.21 为对称矩阵示意图，其中左图为矩阵的所有数据元素，右图为需要存储的数据元素。

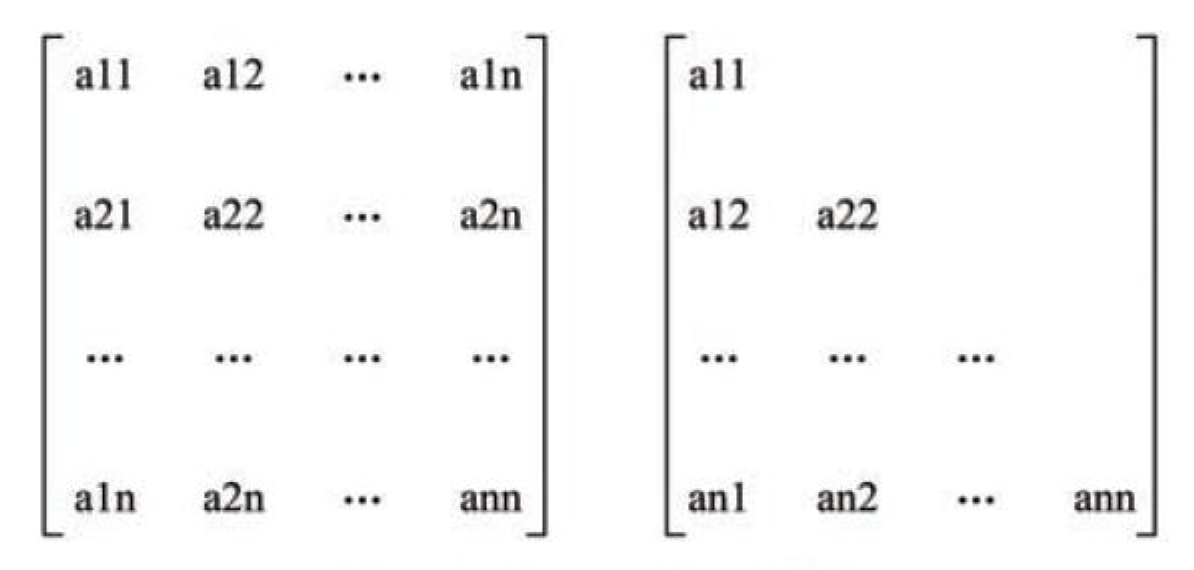


图 2.21　对称矩阵示意图

正常存储需要 n^2 个结点，压缩存储后减少了将近一半的存储量。

对称矩阵的数据元素 $a_{i+1\ j+1}$ 通过二维数组 a[n][n]以行优先的方式压缩存储，其对应数组元素 a[i][j]的地址计算公式如下（ i∈[0,n−1]，j∈[0,n−1] ）。

```
a[i][j].addr= a[0][0].addr+(i*(i+1)/2+j)*sizeof(eleType)       i≥j
a[i][j].addr= a[0][0].addr+(j*(j+1)/2+i)*sizeof(eleType)       i<j
```

i≥j 时，a[i][j]在一维存储空间的相对位置为第 i*(i+1)/2+j 个结点；i<j 时，a[i][j]在一维存储空间的相对位置为第 j*(j+1)/2+i 个结点。

列优先方式存放时的公式请考生自行推算。

2.6.2　三角矩阵

n 阶三角矩阵有上三角矩阵和下三角矩阵之分，分别表示主对角线以下或以上的元素值相同，其他数据元素不同的矩阵。图 2.22 为上三角矩阵示意图，其中左图为上三角矩阵，右图为上三角矩阵需要存储的数据元素，其中左下方 n(n−1)/2 个具有相同值 m 的数据元素在存储时仅占 1 个数组元素空间。

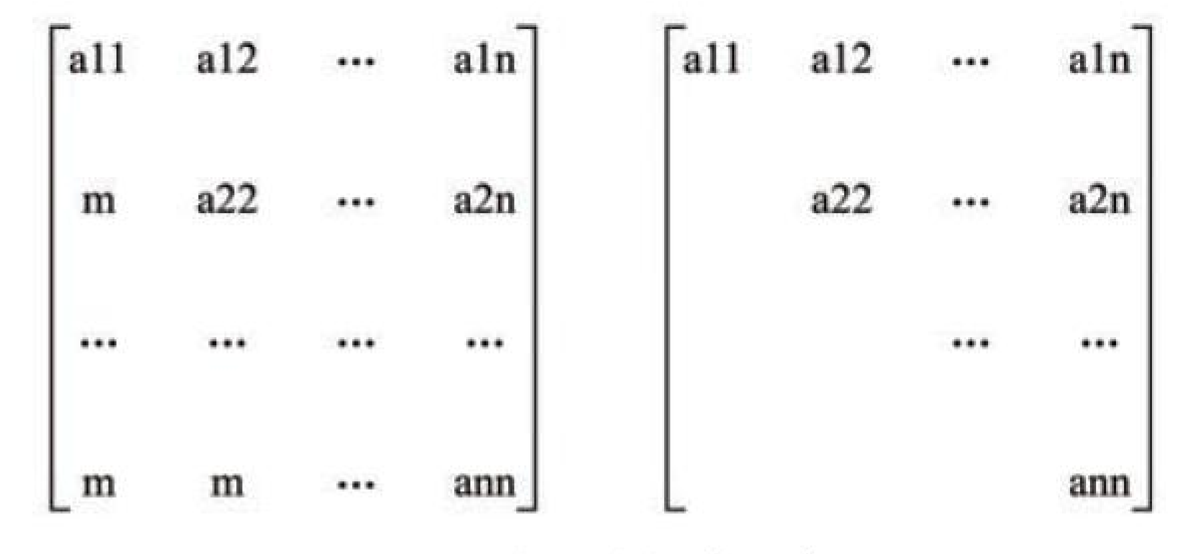


图 2.22　上三角矩阵示意图

图 2.23 为下三角矩阵示意图，其中左图为下三角矩阵，右图为下三角矩阵需要存储的数据元素，其中右上方 n(n−1)/2 个具有相同值 m 的数据元素在存储时仅占 1 个数组元素空间。

$$\begin{bmatrix} a11 & m & \cdots & m \\ a21 & a22 & \cdots & m \\ \cdots & \cdots & \cdots & \cdots \\ an1 & an2 & \cdots & ann \end{bmatrix} \quad \begin{bmatrix} a11 & & & \\ a21 & a22 & & \\ \cdots & \cdots & \cdots & \\ an1 & an2 & \cdots & ann \end{bmatrix}$$

图 2.23　下三角矩阵示意图

n*n 个元素需要的存储结点即数组元素的个数为：

$$1+(1+2+\cdots+n)=1+n(n+1)/2$$

比对称矩阵的压缩存储多一个元素。正常存储需要 n^2 个结点，压缩存储减少了将近一半的存储量。

1．上三角矩阵存储数据元素对应数组下标的计算

上三角矩阵 a_{n*n} 的数据元素 $a_{i+1\ j+1}$ 通过二维数组 a[n][n]以行优先方式压缩存储，其对应数组元素为 a[i][j]。

存储时，首先保存上三角的所有元素，最后保存等值元素 m。

（1）上三角矩阵（i≤j）的所有元素 a[i][j]的下标计算。

数组的第 0 行需要存储 n 个元素，第 1 行需要存储 n−1 个元素……第 i 行需要存储 n−i 个元素。

数组的第 i 行前面共 i−1 行，需要存储的数据元素个数如下：

$$n+(n-1)+(n-2)+\cdots+(n-i+1)= i*(2*n-i+1)/2$$

数组的第 i 行在 a[i][j]前面的数据元素有 j−i 个。即 a[i][j]之前需要存储的数据元素总数为 i*(2*n−i+1)/2+j−i，对应一维存储空间的数组下标范围为 0～i*(2*n−i+1)/2+j−i−1。所以 a[i][j] 在一维存储空间的数组下标为 i*(2*n−i+1)/2+j−i。其地址计算公式如下(i∈[0,n−1]，j∈[0,n−1])：

$$a[i][j].addr= a[0][0].addr+(i*(2*n-i+1)/2+j-i)*sizeof(eleType) \quad i\leqslant j$$

（2）矩阵右下方（i>j）等值元素 m 的下标计算。

上三角的数据元素个数同对称矩阵，为 n*(n+1)/2 个，对应一维存储空间的数组下标范围为 0～n*(n+1)/2−1。所以等值元素 m 的数组下标为 n*(n+1)/2。

$$a[i][j].addr= a[0][0].addr+(n*(n+1)/2)*sizeof(eleType) \quad i>j$$

2．下三角矩阵存储数据元素对应数组下标的计算

下三角矩阵的数据元素 $a_{i+1\ j+1}$ 通过二维数组 a[n][n]以行优先方式压缩存储，其对应数组元素为 a[i][j]。

存储时，首先保存下三角的所有元素，最后保存等值元素 m。

（1）下三角（i≥j）的所有元素对应的 a[i][j]的下标计算。

数组的第 0 行需要存储 1 个元素，第 1 行需要存储 2 个元素……第 i 行需要存储 i+1

个元素。

数组的第 i 行前面共 i-1 行，需要存储的数据元素个数如下：

$$1+2+\cdots+i= i*(i+1)/2$$

数组的第 i 行在 a[i][j]前面的数组元素有 j 个。即 a[i][j]之前需要存储的数据元素总数为 i*(i+1)/2+j，对应一维存储空间的数组下标范围为 0～i*(i+1)/2+j−1。所以 a[i][j] 在一维存储空间的数组下标为 i*(i+1)/2+j。其地址计算公式如下(i∈[0,n−1]，j∈[0,n−1])：

a[i][j].addr= a[0][0].addr+(i*(i+1)/2+j)*sizeof(eleType)　i≥j

（2）矩阵左上方（i<j）等值元素 m 的下标计算。

下三角的数据元素个数同对称矩阵，为 n*(n+1)/2 个，对应一维存储空间的数组下标范围为 0～n*(n+1)/2−1。所以等值元素 m 的数组下标为 n*(n+1)/2。

a[i][j].addr= a[0][0].addr+(n*(n+1)/2)*sizeof(eleType)　i<j

列优先方式存放时的公式请考生自行推算。

2.6.3 对角矩阵

对角矩阵的所有非零元集中在以主对角线为中心的带状区域，如图 2.24 所示。

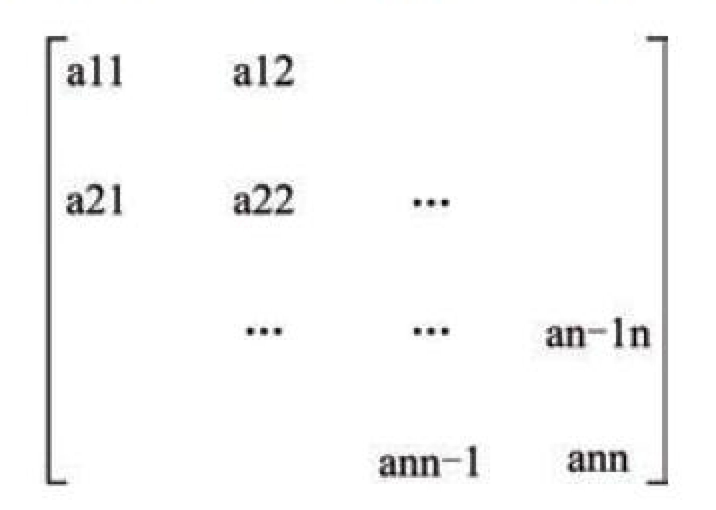


图 2.24　对角矩阵示意图

对角矩阵的非零数据元素 $a_{i+1\ j+1}$ 通过二维数组 a[n][n]以行优先方式压缩存储，其对应数组元素为 a[i][j]，其存储地址如下。

数组的第 0 行需要存储 2 个元素，第 1 行～第 n−2 行需要存储 3 个元素，第 n−1 行需要存储 2 个元素。所有需要存储的数据元素个数为 2+3*(n−2)+2=3n−2。

非 0 元素 a[i][j] 在一维存储空间的地址计算公式如下(i∈[0,n−1]，j∈[0,n−1]，i==j+1 或 i==j 或 i==j−1)：

a[i][j].addr= a[0][0].addr+ j*sizeof(eleType)　i=0

a[i][j].addr= a[0][0].addr+(2+3*(n-2) +(j-(n-2)))*sizeof(eleType)

= a[0][0].addr+(2*n+j-2)*sizeof(eleType)　i=n-1

a[i][j].addr= a[0][0].addr+(2*i+j)*sizeof(eleType)　i!=0, i!=n-1

其中 i=j+1 表示主对角线下方的非零元素，为第 i 行的首个非零元素（i!=0）；i=j 表示主对角线上的非零元素；i=j−1 表示主对角线上方的非零元素，为第 i 行的最后一个非零元

素（i!= n−1）。

非零元素 a[i][j] 在一维存储空间的数组下标 k 的取值如下。

```
k=j;                    // i=0
k=2*n+j-2;              // i=n-1
k=2*i+j;                // i!=0, i!=n-1
```

2.6.4 稀疏矩阵

1. 定义

一个 n×n 矩阵中，设非 0 元素的个数为 t，则 0 元素个数为 n×（n−t）。若 t 远小于 n*n−t)，且非 0 元素的分布没规律，这样的矩阵称为稀疏矩阵，如图 2.25 所示。

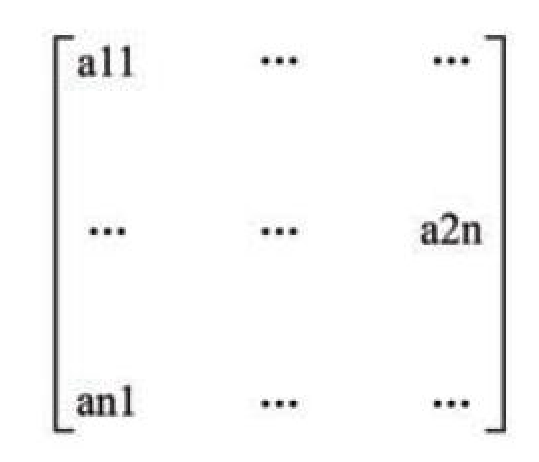


图 2.25　稀疏矩阵示意图

设 δ=t/(n*n)，一般当 δ≤0.05 时为稀疏矩阵，δ 为稀疏因子。

2. 存储结构及应用

为合理利用存储空间，稀疏矩阵一般采用压缩存储。由于非 0 元素的分布没有规律，所以存储数据元素的同时需要存储其位置信息。常见的稀疏矩阵压缩存储方式有三元组和十字链表。

1）三元组

（1）三元组的概念

稀疏矩阵中的每个非 0 元素用三元组(i, j, aij)唯一确定，其中，i 表示元素 aij 所在的行号，j 表示元素 aij 所在的列号。所有非 0 元素的三元组存放于一个一维数组中。如图 2.26 所示，左图为一稀疏矩阵，右图为其三元组存储结构示意图，设三元组存放于数组 a 中。

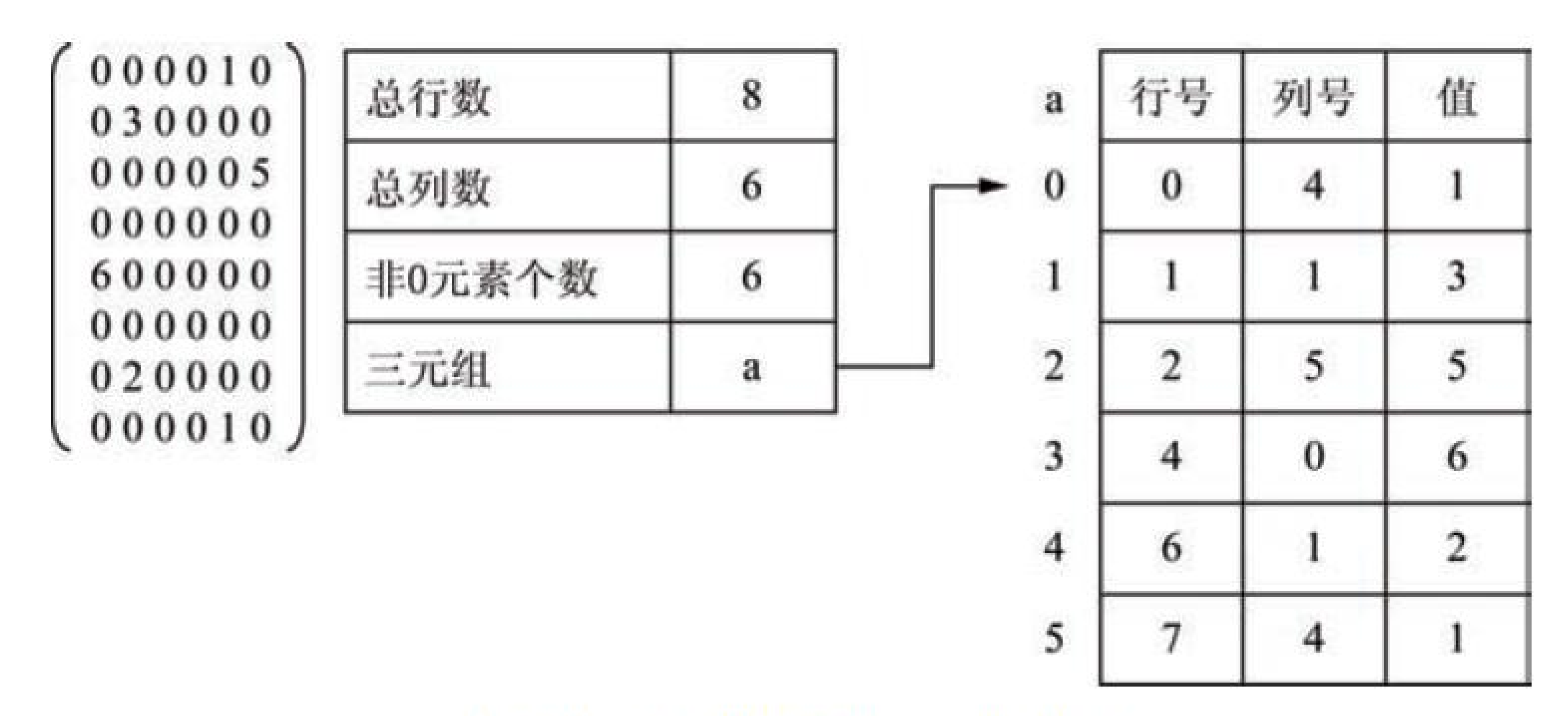


图 2.26　稀疏矩阵的三元组存储

（2）三元组的存储结构

稀疏矩阵的三元组存储需要定义两个类型，一个为三元组结点类型 spNode，存储非 0 元素的行号、列号以及值；另外一个为三元组表类型 spMatrix，存储稀疏矩阵的总行数、总列数、非 0 元素个数，以及对应的三元组。

```
#define spMAX  100
typedef  struct
  {  unsigned  i,j;        // 非 0 元素的行号 i、列号 j
     eleType  x;           // 非 0 元素的值
  }spNode;                 // 三元组结点类型
typedef  struct
 {  unsigned  mu,nu,tu;   // 稀疏矩阵的总行数 mu、总列数 nu、非 0 元素个数 tu
    spNode  data[spMAX];  // 三元组
 } spMatrix;               // 三元组表存储类型
```

（3）三元组的基本操作

① 稀疏矩阵的三元组创建。

设稀疏矩阵为 a[M][N]，对应三元组表为 sqA，则创建 a[M][N]的三元组表 sqA 的算法描述如下。

```
#define M 10000
#define N 20000
void creatSpA( spMatrix *sqA, eleType a[M][N]){
    unsigned i,j;
    sqA->mu=M;
    sqA->nu=N;
    sqA->tu=0;
    for(i=0;i<M;i++)
        for(j=0;j<N;i++)
            if( a[i][j]!=0 ) {
                sqA->data[sqA->tu].i= i;
                sqA->data[sqA->tu].j= j;
                sqA->data[sqA->tu].x= a[i][j];
                sqA->tu++;
            }
}
```

② 稀疏矩阵的转置。

求稀疏矩阵 A 的转置矩阵 B，设 A 对应三元组表为 sqA，B 对应三元组表为 sqB。快速方法为先计算每个非零元转置后在三元组中的存储位置，然后遍历三元组，将非 0 元素放入转置后的位置。

- 计算每个非 0 元素转置后在三元组中的存储位置。

设 num[col]表示第 col 列非 0 元素的个数，cpot[col]表示第 col 列中第一个非 0 元素在转置矩阵对应的三元组中的位置。则：

```
cpot[0]=0
cpot[col]=cpot[col-1]+num[col-1]   col>0
```

算法描述如下。

```
unsigned col;
unsigned cpot[sqMAX]={0};
unsigned num[sqMAX]={0};
               // 第一次遍历稀疏矩阵 A 的三元组表，求每列的非 0 元素个数
for(unsigned k=0;k<sqA.tu;k++)
    num[ sqA.data[k].j ]++;
```

```
            // 第二次遍历稀疏矩阵 A 的三元组表，求每列的第一个非 0 元素在稀疏
            // 矩阵 B 的三元表中的位置
for(unsigned k=0;k<sqA.nu;k++)
   cpot[k]= cpot[k-1]+num[k-1];
```

● 遍历三元组，将非 0 元素放入转置后的位置。

```
sqB.mu=sqA.nu;
sqB.nu=sqA.mu;
sqB.tu=sqA.tu;
for(unsigned k=0;k<sqA.tu;k++){
   sqB.data[cpot[sqA.data[k].i]].j=sqA.data[k].i;
   sqB.data[cpot[sqA.data[k].i]].i=sqA.data[k].j;
   sqB.data[cpot[sqA.data[k].i]].x=sqA.data[k].x;
   cpot[i]++;
}
```

求稀疏矩阵 A 的转置矩阵 B 的算法如下。

```
void Atrans2B( spMatrix sqA, spMatrix *sqB ){
   unsigned col;
   unsigned cpot[sqMAX]={0};
   unsigned num[sqMAX]={0};
   for(unsigned k=0;k<sqA.tu;k++)
       num[sqA.data[k].j]++;
   for(unsigned k=0;k<sqA.nu;k++)
       cpot[k]= cpot[k-1]+num[k-1];
   sqB->mu=sqA.nu;
   sqB->nu=sqA.mu;
   sqB->tu=sqA.tu;
   for(unsigned k=0;k<sqA.tu;k++){
      sqB->data[cpot[sqA.data[k].i]].j=sqA.data[k].i;
      sqB->data[cpot[sqA.data[k].i]].i=sqA.data[k].j;
      sqB->data[cpot[sqA.data[k].i]].x=sqA.data[k].x;
      cpot[sqA.data[k].i]++;
   }
}
```

2）十字链表

（1）十字链表的概念

当稀疏矩阵中非 0 元素的个数和位置在操作过程中变化较大时，用十字链表作存储结构的算法实现具有较高的效率。每个非 0 元素用含 5 个域的结点表示，其中 i、j 和 x 的含义和三元组表示相同，分别表示该非 0 元素所在行号、列号和值；right 域用来链接同一行的下一个非 0 元素，down 域用来链接同列的下一个非 0 元素，如图 2.27 所示。



图 2.27　十字链表非零元结点结构

稀疏矩阵中同一行的非 0 元素通过向右的 right 指针链接为一个带表头结点的链表；

同一列的非 0 元素通过向下的 down 指针链接为一个带表头结点的链表。因此，每个非 0 元素既是第 i 行链表中的一个结点，又是第 j 列链表中的一个结点，类似十字交叉，故称十字链表。

十字链表结构如图 2.28 所示。



图 2.28　十字链表结构

图 2.29 中，左图为一稀疏矩阵，右图为其十字链表存储结构示意图。

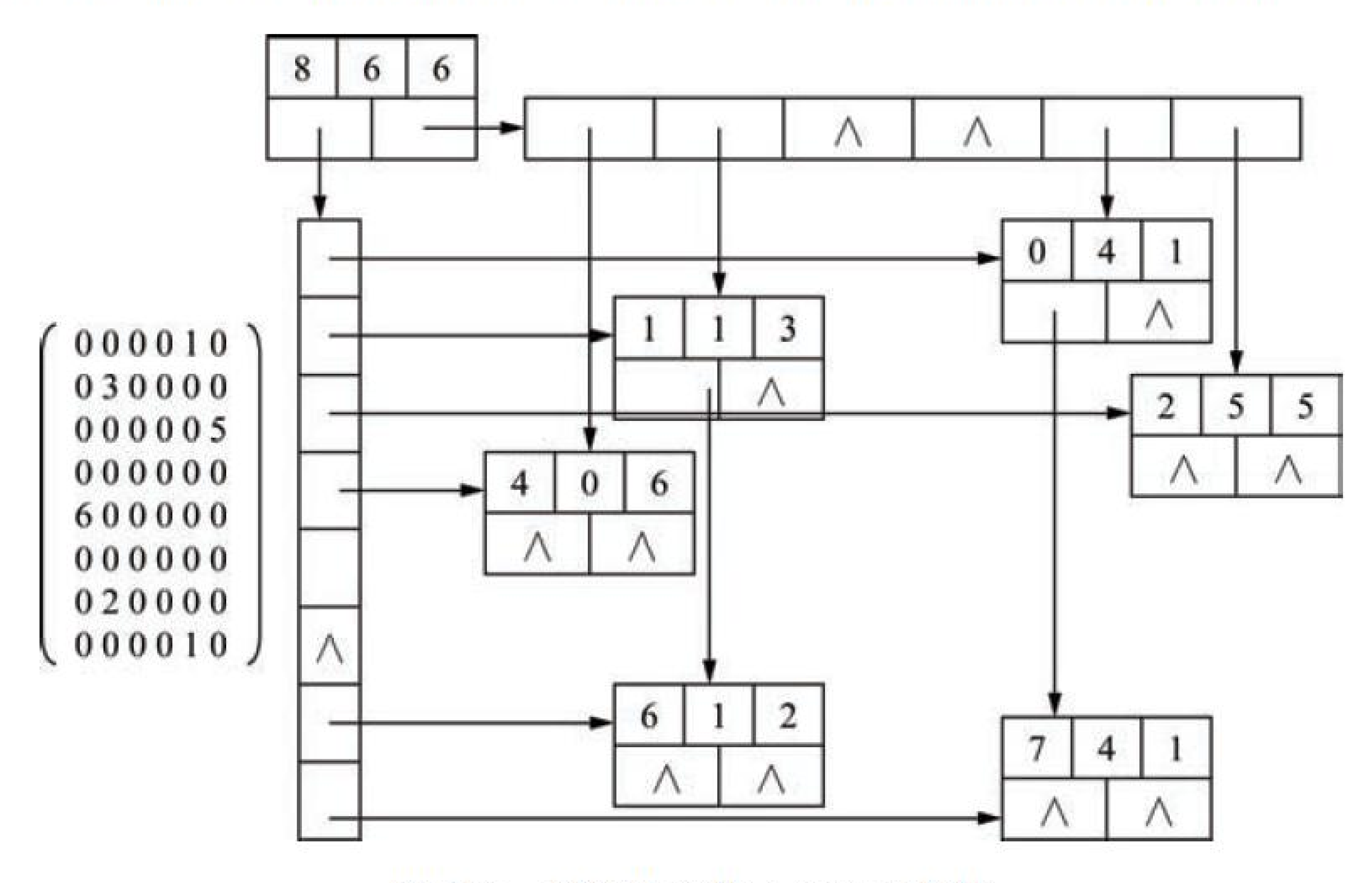


图 2.29　稀疏矩阵的十字链表存储

（2）十字链表的存储结构

稀疏矩阵的十字链表存储需要定义两个类型，一个为结点类型 croNode，存储非 0 元素的行号、列号、非 0 元素的值、同行下一个非 0 元素结点地址、同列下一个非 0 元素结点地址；另外一个为十字链表存储类型 croLinkList，存储稀疏矩阵的总行数、总列数、非 0 元素个数，以及分别指向行指针数组和列指针数组的指针。

```
typedef  struct cNode{
    unsigned  i,j;              // 非 0 元素的行号 i、列号 j
    eleType  x;                 // 非 0 元素的值
    struct cNode *down, *right;
 }croNode, *cLink;              // 十字链表结点类型
typedef  struct{
   unsigned  mu,nu,tu;          // 稀疏矩阵的总行数 mu、总列数 nu、非 0 元素个数 tu
   cLink  cHead[N], rHead[M];   // 指向行数组 rHead 和列数组 cHead
 }croLinkList;                  // 十字链表存储类型
```

2.7 串

串是数据元素为字符类型的特殊线性表，通常作为整体参与处理。

2.7.1 基本概念

（1）串：又称字符串，由用双引号括起的 0 个或多个字符组成的有限字符序列。如 str="$s_1s_2\cdots s_n$"，其中 s_i 为字符类型，str 称为串名，n 称为串长，n=0 称为空串。

（2）子串：字符串（主串）中任意多个连续字符所组成的子序列，串中第 1 个字符在主串中的位置，称为该子串在主串中的位置。

2.7.2 存储结构

串的存储结构有以下两种。

1. 顺序存储

顺序存储又称顺序串，用一组地址连续的存储单元存储字符串的存储方式，通常高级语言选择用数组。定义如下。

```
#define  MAX_STR_LEN  1000              // 最大串长
typedef  struct{
   char  s[MAX_STR_LEN +1];             // 多申请一个空间存放字符串结束标记
   int    length;                       // 串的实际字符个数
}seqString;
```

顺序存储的优点是简单方便。缺点是 MAX_STR_LEN 尽可能定义得大一些，否则应用中容易益出，丢失有效字符。但过大的 MAX_STR_LEN 会导致存储空间浪费。

2. 链式存储

链式存储是动态分配存储空间，定义如下。

```
#define  MAX_STR_LEN  1000              // 最大串长
typedef  struct{
       char *ch;
       int length;                      // 串的实际字符个数
}linkString;
定义变量：linkString  linkS;
申请空间：linkS.ch= (char *)malloc(MAX_STR_LEN+1);
释放空间：fee(linkS.ch);
```

2.7.3 基本操作

本节串的算法以链式存储结构为例。

1．初始化串

给字符串 str 申请空间，并赋初值。

```
void stringInit( linkString  *strD, int  len){
       strD->ch=(char *)malloc(len+1);  // 申请 len+1 个元素的存储空间
                                      // 第 0～len-1 个存储单元存放字符串，第 len
                                      // 个（最后一个）存储单元存放字符串结束
                                      // 标记'\0'
       if(strD->ch==NULL)             // 申请空间失败则提示出错并退出
           exit("申请空间失败，退出");
       for(int  i=0;i<=len;i++)
           strD->ch[i]='\0';          // 将字符串的所有字符初始化为 ASCII 码
                                      // 值为 0 的字符
       strD->length=0;                // 串长置为 0
}
```

2．求串长

求字符串 str 的长度，即串中包含的有效字符个数。

```
int  stringLength ( linkString  *str){
   return  str->length;               // 返回串长度
}
```

3．拷贝赋值

将字符串 str 赋值给串 strD。

```
void stringCopy ( linkString  *strD,  linkString   *str ){
   int len=stringLength (str);        // 字符串 str 的长度赋给 len
   if( !len ){                        // 如果串长为 0，则字符串 strD 置空串
       strD->ch=NULL;
       strD->length=0;
   }else{                             // 如果字符串 str 不是空串，依次赋值
       strD->ch=(char *)malloc(len+1);    // 申请 len+1 个元素的存储空间
                                      // 第 0～len-1 个存储单元存放字符串,第 len
                                      // 个（最后一个）存储单元存放字符串结束
                                      // 标记'\0'
       if(strD->ch==NULL)             // 申请空间失败则提示出错并退出
           exit("申请空间失败，退出");
       for(int  i=0;i<=len;i++)
           strD->ch[i]=str->ch[i];
       strD->length=len;              // 修改串长
   }
}
```

4．插入

将字符串 strT 插入到串 strD 中 index 开始的位置。

```
void stringInsert( linkString  *strD,  linkString  *strT ,int  index ){
   if( index<0  || index> stringLength (strT)+1)  // 插入位置不合法
       exit("插入位置不合法，退出");
   int lenT=stringLength (strT);              // 字符串 strT 的长度赋给 lenT
   int lenD=stringLength (strD);              // 字符串 strD 的长度赋给 lenD
```

```
    int i;
    strD->ch=(char *)realloc(strD->ch, lenT+lenD+1);
                     // 申请首地址为strD->ch、lenT+lenD+1个数据元素所需的存储空间
                     // 多申请的1个单元存放字符串结束标记'\0'
                     //  strD仍然保留原字符串
    if(strD->ch==NULL)                      // 申请空间失败则提示出错并退出
        exit("申请空间失败，退出");
    for(i=lenD;i>=index;i--)
                      // 原串位于区间 [lenD，index ]的字符依次后移lenT个字符位置
        strD->ch[i+lenT]= strD->ch[i];
    for(i=0;i<lenT;i++)                     // 将strT所含字符串插入strD第
                                            // index~index+ lenT-1区间
        strD->ch[i+index]= strT->ch[i];
    strD->length=lenT+lenD;                 // 修改串长
}
```

5．删除若干字符

删除字符串 strD 中 index 位置开始的 n 个字符。

```
void stringDeleteN( linkString  *strD, int  index,int  n ){
    int lenD=stringLength (strD);          // 字符串strD的长度赋给lenD
    if( index<0  || index>=lendD)          // 删除位置不合法
        exit("删除位置不合法，退出");
    if( index+n-1>=lenD )                  // 从index位置开始不够n个字符
        exit("从index位置开始不够n个字符，退出");
    for( int  i=index;i<index+n;i++)
                                           // 原串位于区间 [index，index+n-1]
                                           // 的n字符依次前移n个字符位置
        strD->ch[i-n]=strD->ch[i];
    strD->length=strD->length - n;         // 修改串长
}
```

6．串删除

删除字符串 strD，释放整个串空间。

```
void stringDelete( linkString  *strD ){
    if(strD->ch!=NULL ){
        free(strD->ch);
        strD->length=0;
    }
}
```

7．串比较

按字典顺序比较字符串 strS 和 strD。

（1）strS > strD，返回 1。

（2）strS == strD，返回 0。

（3）strS < strD，返回-1。

```
int  stringCompare(linkString  strS[ ],linkString  strD[ ] ){
    for( int i=0; i< strS.length && i< strD.length; i++){
        if( strS.ch[i] == strD.ch[i] )
            continue;
```

```
        if( strS.ch[i] > strD.ch[i] )
            return 1;
        if( strS.ch[i] < strD.ch[i] )
            return -1;
    }
    if( strS.length == strD.length )
        return 0;
    if( strS.length > strD.length )
        return 1;
    if( strS.length < strD.length )
        return -1;
}
```

8. 串连接

将字符串 strS 连接到串 strD 的后面。

```
void stringConcat( linkString  *strD, linkString  *strS ){
    int lenS=stringLength (strS);          // 字符串 strS 的长度赋给 lenS
    int lenD=stringLength (strD);          // 字符串 strD 的长度赋给 lenD
    strD->ch=(char *)realloc(strD->ch, lenS+lenD+1);
      // 申请首地址为 strD->ch、lenS+lenD+1 个数据元素所需的存储空间
      // 多申请的 1 个单元存放字符串结束标记'\0'
      // strD 仍然保留原字符串
    if(strD->ch==NULL)                     // 申请空间失败则提示出错并退出
        exit("申请空间失败，退出");
    for(i=0;i<=lenS;i++)                   // strS 的字符依次连接到串 strD 的后
                                           // 面，包括字符串结束标记
      strD->ch[i+lenD]=strS->ch[i];
    strD->length=lenS+lenD;                // 修改串长
}
```

2.7.4　模式匹配

子串 strT（模式）在主串 strS 中的定位称为串的模式匹配。功能为在 strS 中查找与子串 strT 完全匹配/相同的子串。若查找成功，则返回模式串 strT 的第一个字符在主串 strS 中出现的位置，否则返回−1。

1. 一般匹配算法

1）算法思想

一般匹配算法的主要思想如下。

（1）主串 strS 的开始位置 i=0，长度为 lenS；strT 的开始位置 j=0 开始，长度为 lenT。

（2）通过 i++、j++依次匹配 strS 的第 i 个字符和 strT 的第 j 个字符。

（3）遇到第一个不匹配的字符时通过 i=i−j+1（上次开始位置的下一个位置）、j=0 进行回溯。

（4）重复步骤（2）、（3）继续匹配，直到 i==lenS - lenT 并且 strD->ch[i] != strT->ch[j]，或 j==lenT。

（5）若 i==lenS，匹配失败。

（6）若 j==lenT，匹配成功，返回 strT 的第一个字符在主串 strS 中出现的位置 i-j+1（即下标+1）。

2）伪代码

```
void stringMatch( linkString  *strS, linkString  *strT ){
    int i=0,j=0;
    while( i<= strS->length - strT->length &&  j< strT->length )
        if( strD->ch[i] == strT->ch[j] )
            i++, j++;
        else if( i!= strS->length - strT->length )
                i=i-j+1, j=0;
            else
                return -1;
    if( j==strT->length )
        return  i-j+1; //返回strT的第一个字符在主串strS中的位置i-j+1(即下标+1)
    else
        return  -1;
}
```

3）示例

（1）最好匹配情况。

如 strD->ch={"123abc"}，strT->ch={"123"}，匹配过程如图 2.30（a）和图 2.30（b）所示。

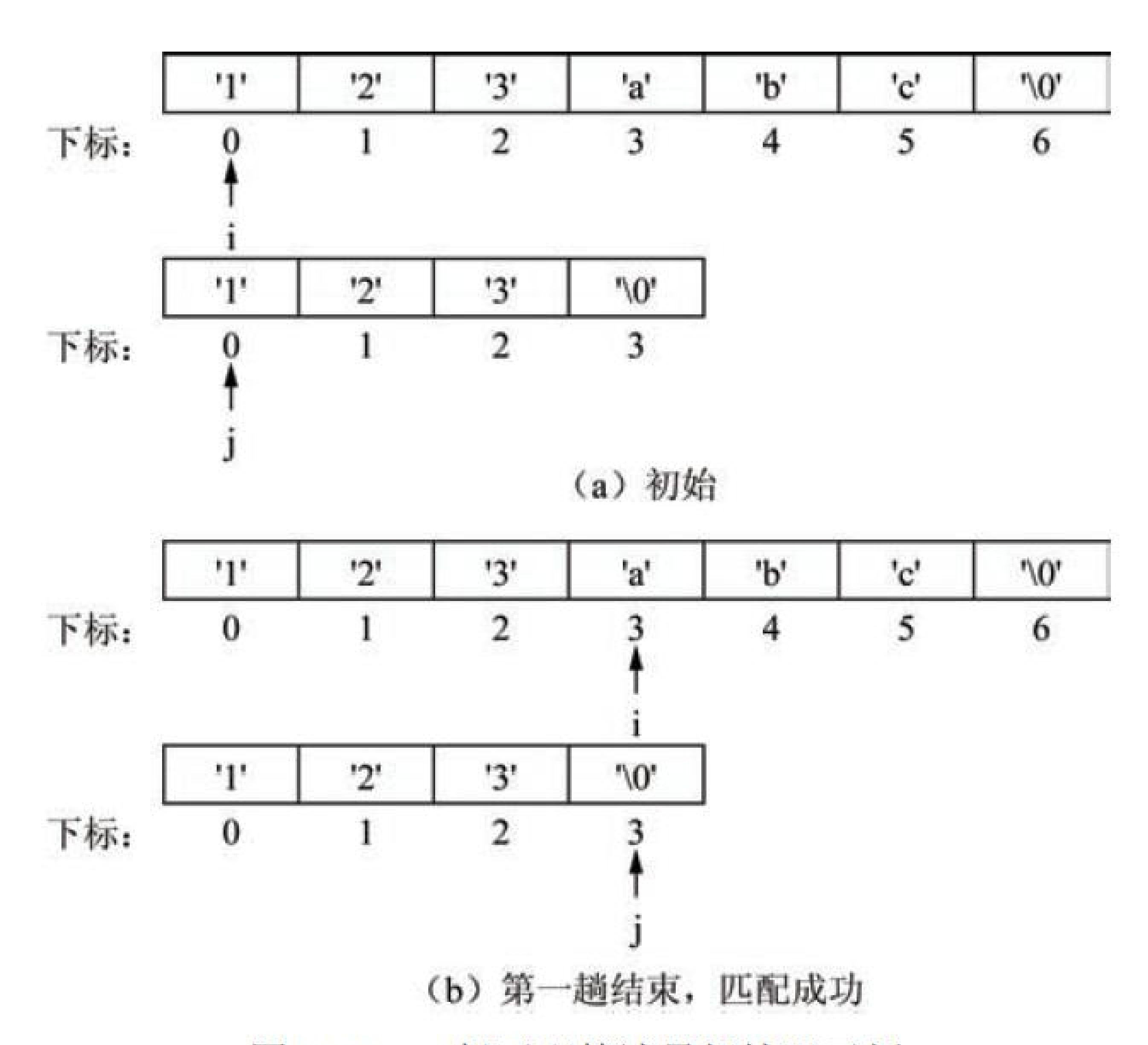
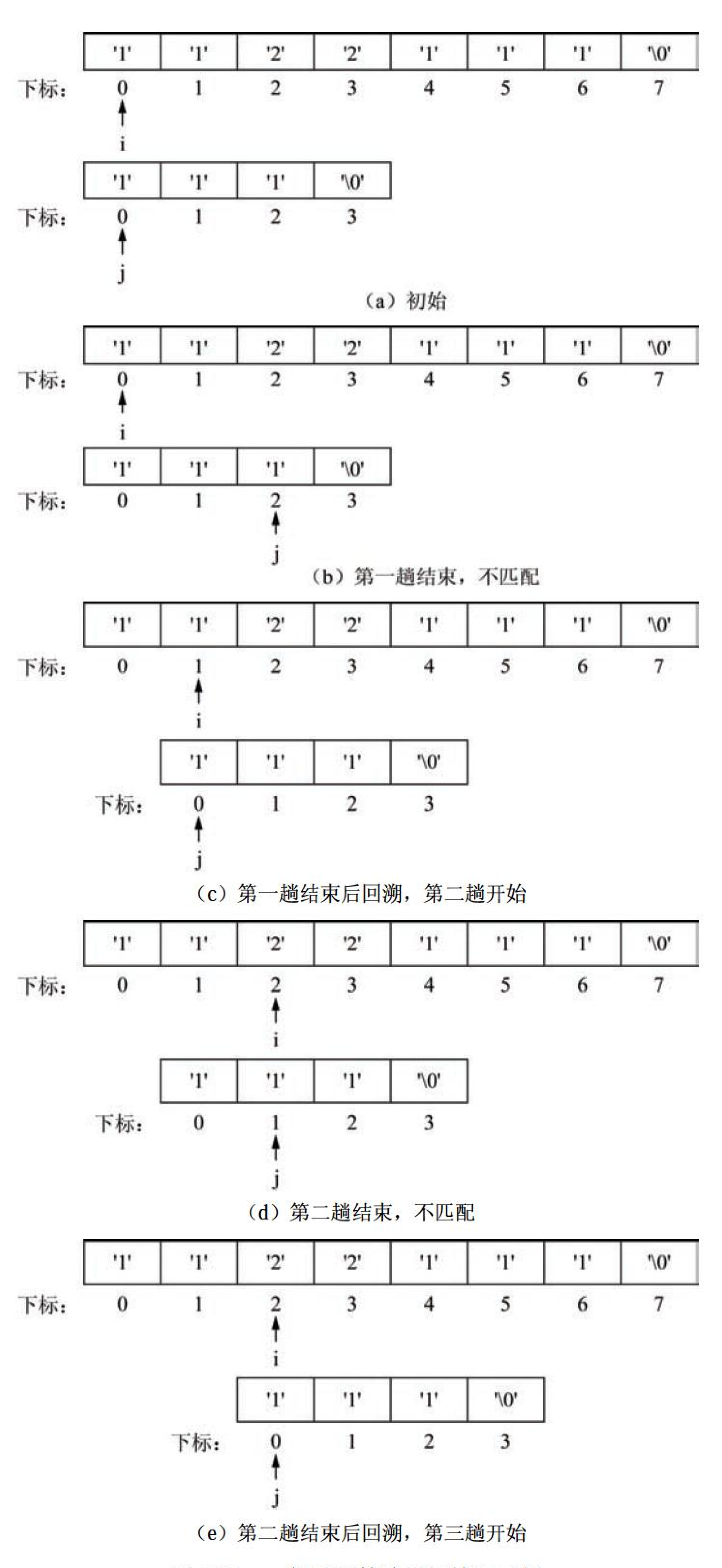


图 2.30　一般匹配算法最好情况示例

（2）最坏匹配情况。

如 strD->ch={"1122111"}，strT->ch={"111"}，匹配过程如图 2.31（a）～图 2.31（h）所示。



（a）初始

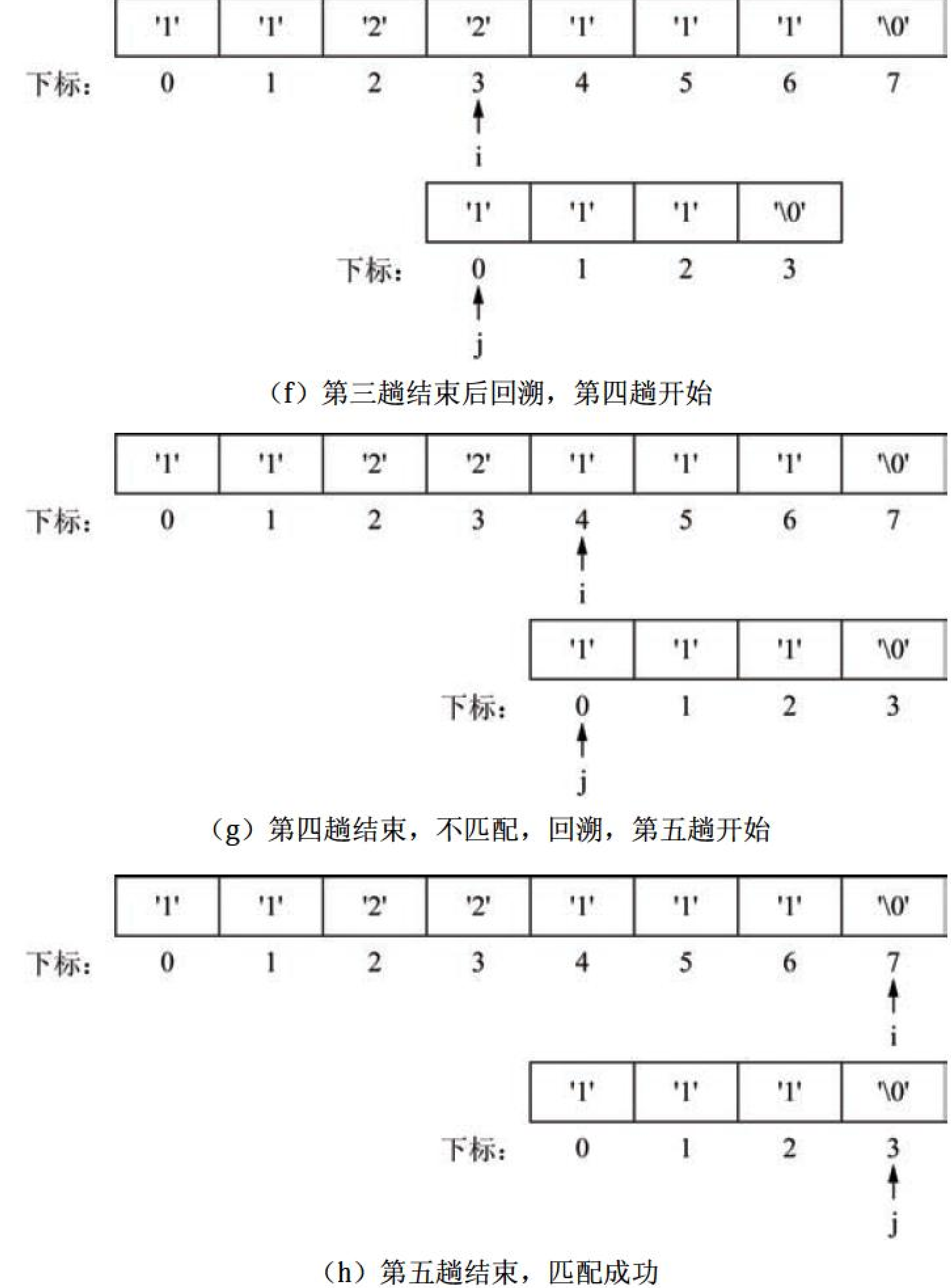
（b）第一趟结束，不匹配

（c）第一趟结束后回溯，第二趟开始

（d）第二趟结束，不匹配

（e）第二趟结束后回溯，第三趟开始

图 2.31　一般匹配算法最坏情况示例



（f）第三趟结束后回溯，第四趟开始

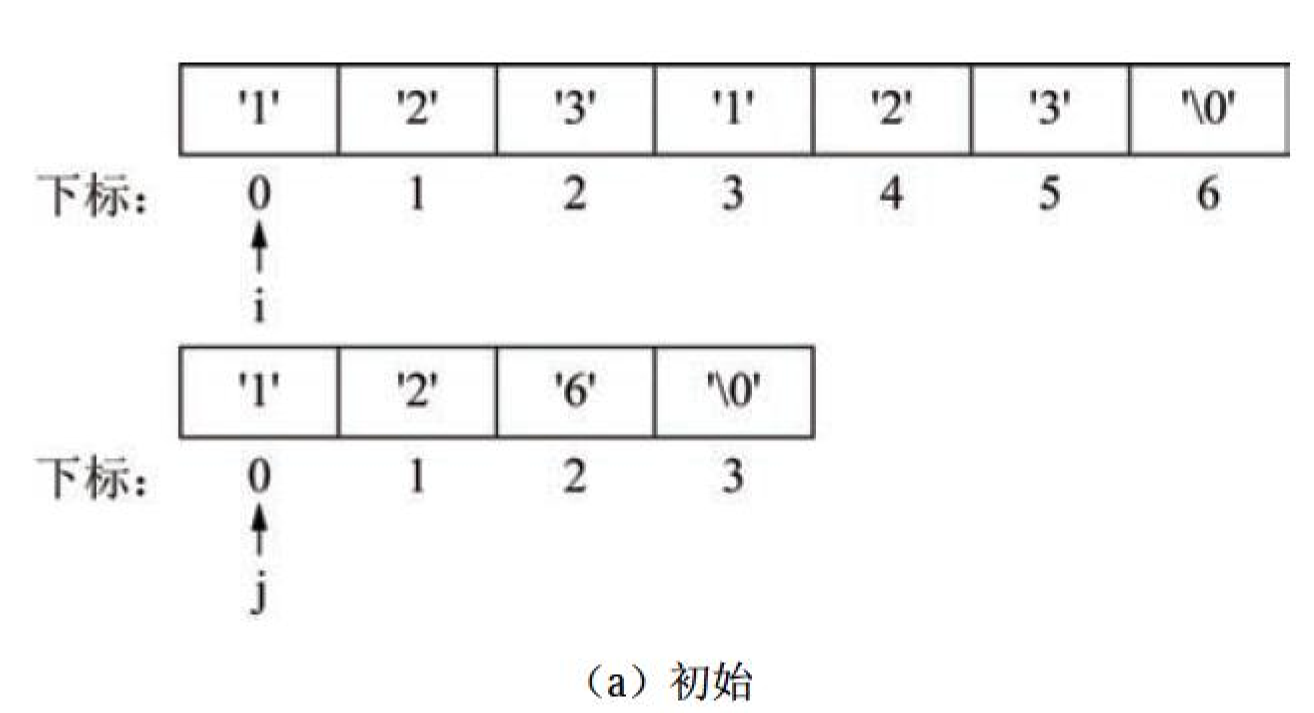
（g）第四趟结束，不匹配，回溯，第五趟开始

（h）第五趟结束，匹配成功

图 2.31（续）

（3）不匹配情况。

如 strD->ch={"123123"}，strT->ch={"126"}，匹配过程如图 2.32（a）～2.32（f）所示。



（a）初始

图 2.32　一般匹配算法不匹配情况示例

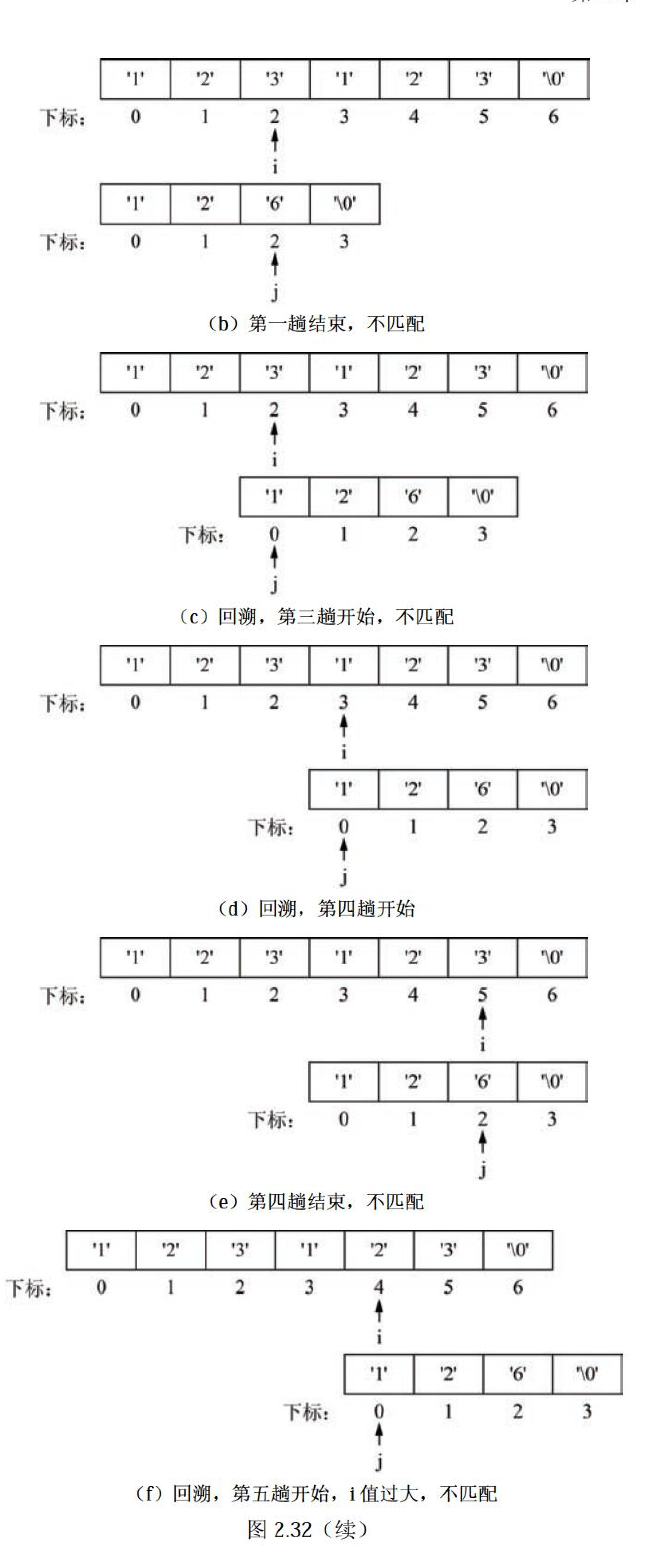

'1' '2' '3' '1' '2' '3' '\0'
下标： 0 1 2 3 4 5 6
i
'1' '2' '6' '\0'
下标： 0 1 2 3
j
（b）第一趟结束，不匹配
'1' '2' '3' '1' '2' '3' '\0'
下标： 0 1 2 3 4 5 6
i
'1' '2' '6' '\0'
下标： 0 1 2 3
j
（c）回溯，第三趟开始，不匹配
'1' '2' '3' '1' '2' '3' '\0'
下标： 0 1 2 3 4 5 6
i
'1' '2' '6' '\0'
下标： 0 1 2 3
j
（d）回溯，第四趟开始
'1' '2' '3' '1' '2' '3' '\0'
下标： 0 1 2 3 4 5 6
i
'1' '2' '6' '\0'
下标： 0 1 2 3
j
（e）第四趟结束，不匹配
'1' '2' '3' '1' '2' '3' '\0'
下标： 0 1 2 3 4 5 6
i
'1' '2' '6' '\0'
下标： 0 1 2 3
j
（f）回溯，第五趟开始，i 值过大，不匹配

图 2.32（续）

4）性能分析

图 2.30～图 2.32 给出了一般字符串匹配算法的两大类情况，说明如下。

（1）匹配成功的情况。

图 2.30 为匹配成功的最好情况，即主串的前 lenT 个字符即为 strT 中字符，故返回值为 str->ch T 的第一个字符在主串 strS->ch 中的位置 1（下标 0 加 1），这种情况的时间复杂度为 O(lenT)。

图 2.31 为匹配成功的最坏情况，即主串的前 k-1 个长度为 lenT-1 的字符均和 strT 中的前 lenT-1 个字符匹配，只有最后一个不匹配；并且主串最后的 lenT 个字符与 strT 匹配。故返回值为 strT 的第一个字符在主串 strS 中的位置 lenS-lenT+1（下标 lenS-lenT 加 1）。这种情况的时间复杂度为 O(lenS*lenT)。

（2）匹配不成功的情况

如图 2.32 所示，其中 i 的值为 4，lenT 的值为 3，i+lenT 的值为 7，主串 strS->ch 中不可能包含子串 strT->ch，算法可结束，时间复杂度为 O(lenS*lenT)。

2．改进的匹配算法——KMP 算法

1）算法思想

分析一般匹配算法可发现，回溯操作部分可以加快，没有必要将主串指针回溯到上次比较开始位置的下一个位置。设子串 strT 包含的有效字符为 strT->ch[0], strT->ch[1], …, strT->ch[i] , …, strT->ch[lenT-1]。

（1）如果当前子串 strT 中待比较的字符为 strT->ch[k]，则 strT->ch[0]～strT->ch[i-1] 已经匹配成功，即 strT->ch[0]～strT->ch[k-2]和 strS->ch[i-k]～strS->ch[i-2]对应位置字符相等。

（2）已经得到的匹配结果为 strT->ch[j-k]～strT->ch[j-2]和 strS->ch[i-k]～strS->ch[i-2]对应位置字符相等。

（3）由（1）和（2）可知，strT->ch[**0**]～strT->ch[k-2]和 strT->ch[**j-k**]～strT->ch[j-2]对应位置字符相等。

（4）一般匹配算法回溯时，i 和 j 均回退，其中 j 回退至 0，从第一个字符重复重叠继续比较，所以时间复杂度高。

（5）利用（3） 的结论，KMP 算法回溯时，i 不动，j 回退至 j-k，不仅使 i 不再回退，而且 j 回退缩短，可有效降低时间复杂度为 O(lens+lenT)。

2）KMP 算法的关键——next 值求解

（1）设 next[j]=k，则 next[j]表示当子串中第 j 个字符与主串中对应字符不匹配时，子串指示 j 回退到的下标。

（2）next 值计算

next[j]=
- 0: j=1，第一个字符不匹配，直接将主串待比较字符后移一个
- max(k|1<k<j 且 strT->ch[1]～strT->ch[k-1] 和 strT->ch[j-k+1]～strT->ch[j-1] 对应位置字符相等)，集合不空
- 1: 其他

如 strT ={"abaabcac" }的 next 值如下。

j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
next[j]	0	1	1	2	2	3	1	2

该方法可以优化，考生在熟悉该计算方法的基础上，可思考更好的求解方法。

3）伪代码

```
void getNext( linkString  *strT, int  next[]) {
    int  next[1]=0,j=1, k=0;
    while (j < strT->length) {
        if ( k== 0 || strT[j] ==strT[k] ){
            ++j;
            ++k;
            next[j] = k;
        } else
            k = next[k];
    }
}
int  KMPMatch( linkString  *strS, linkString  *strT) {
    int  i=1, j=1 ;
    int  next[]=(int *)malloc((strT->string+1)*sizeof(int));
    getNext(strT, next);
    while (i <= strS->length  && j <= strT->length) {
        if ( j== 0 || strS[i] ==strT[j] ){
            ++j;
            ++k;
        } else
            j = next[j];
    }
    if(j > strT->length )
        return  i- strT->length;
    else
        return 0;
}
```

2.8　综 合 应 用

下面举例说明线性表的实际应用。

2.8.1　两栈共享空间

由于栈只有一端的地址可以随出入栈操作动态变化，另外一端不变的特点，让两个栈共享一个连续空间，两个栈的栈底分别位于连续空间的两端，做入栈操作时栈顶向连

续空间的中间位置移动，出栈操作时栈顶向连续空间的两端移动，如图 2.33 所示。

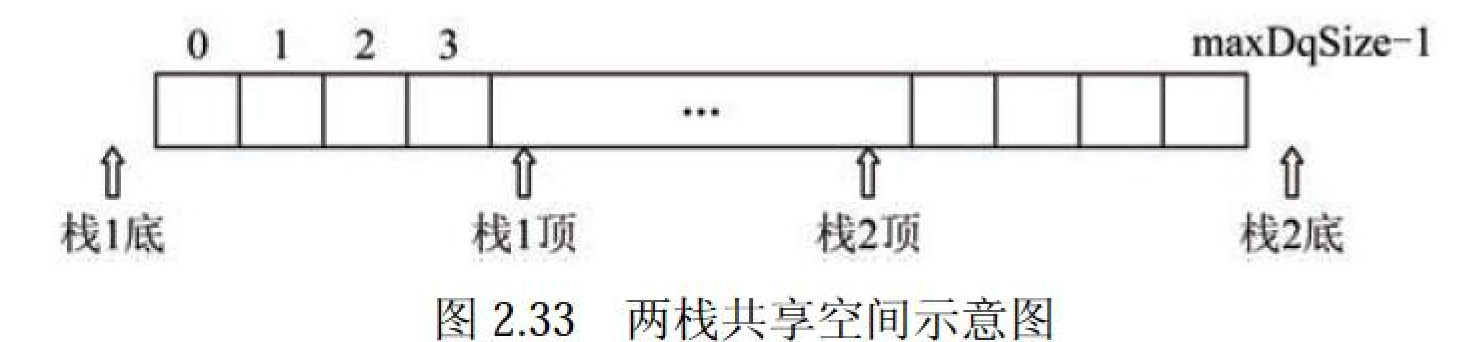


图 2.33　两栈共享空间示意图

两栈共享空间的优点如下。

（1）只要两个栈的数据元素个数之和不大于连续空间的总容量就可进行入栈操作。

（2）每个栈的栈满条件为两个栈顶相等。

（3）只要满足入栈条件，每个堆栈的最大长度可变。

伪代码的实现如下。

```
#define maxDqSize 1000
typedef struct{
    eleType Stack[maxDqSize];
    int top[2];                       // top[0]和top[1]分别为两个栈的栈顶指示器
}DqStack;
void InitStack(DqStack *S) {
    S->top[0]=-1;
    S->top[1]= maxSize;
}
int push(DqStack *S, eleType x, int i){
    if(S->top[0]+1==S->top[1])    // 共享的栈空间已满
        return -1;
    switch(i){
        case 0: S->top[0]++;
              S->Stack[S->top[0]]=x;
              break;
        case 1: S->top[1]--;
              S->Stack[S->top[1]]=x;
              break;
        default: return -1;
    }
    return 1;
}
int pop(DqStack *S, eleType *x, int i){
    switch(i){
        case 0: if(S->top[0]==-1)
                return -1;
              *x=S->Stack[S->top[0]];
              S->top[0]--;
              break;
        case 1: if(S->top[1]==M)
                return -1;
              *x=S->Stack[S->top[1]];
              S->top[1]++;
              break;
        default: return -1;
    }
    return 1;
}
int emptyDqStack(DqStack *S){
    switch(i){
```

```
        case 0: return S->top[0]==-1;
        case 1: return S->top[1]== maxDqSize;
    }
    return 1;
}
int fullDqStack(DqStack *S){
    return S->top[0]+1==S->top[1];
}
```

2.8.2　多项式求和

多项式运算是线性表的典型应用，本节以多项式求和为例介绍其算法及其实现。数学中的多项式为如下表达式：

$$p = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

其中，p 称为 n 项多项式，其中 a_i 为系数， x 为自变量，i 为指数（$0 \leqslant i \leqslant n$）。本节以带头结点的链表存储多项式，链表中除头结点外的每个结点对应多项式的一项，存储该项的系数和指数，其结点结构定义如下。

```
typedef  struct poly {
    int  coef ;                // 变量的系数
    int  exp ;                 // 变量的指数
    struct  poly  *next;       // 指到下一结点的指针
} Lpoly;
```

每个多项式由多个结点构成，高指数项（高次幂）的结点在链表头部，低指数项（低次幂）的结点在链表后部。A、B 两个多项式的链表结构如图 2.34 所示。

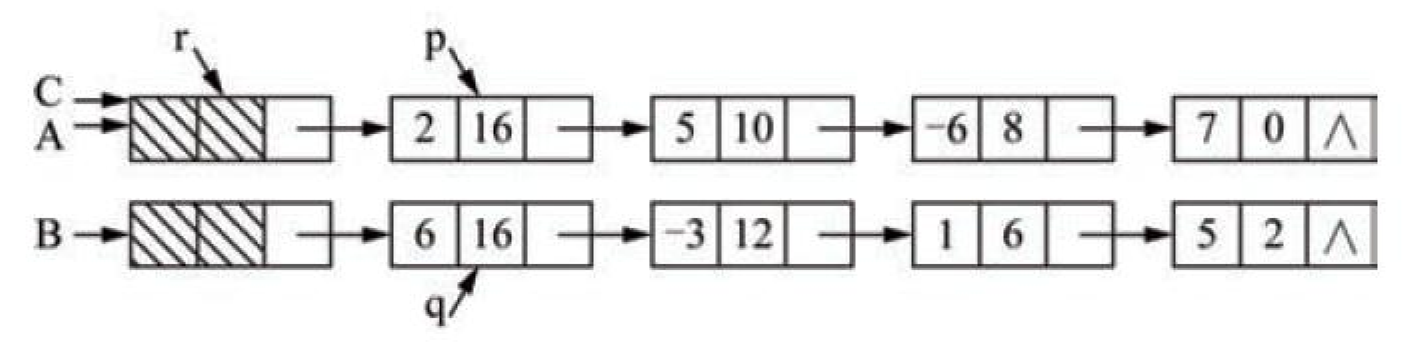


图 2.34　多项式求和初始化

进行加法运算，首先设置 p、q 两个指针变量分别指向 A、B 两个链表的第一个数据元素结点。然后对 p、q 两个结点的指数域进行比较，指数相同则系数相加，连入 C 链表；指数不同，则将指数较大的结点连入 C 链表。

1. 具体算法如下

（1）设 p、q 分别指向 A、B 中某一结点，初值为相应链表的第一数据结点。

（2）C 为 A、B 和的最终链表表头指针，初值为 A（充分利用现有结点，节省存储资源）。

（3）比较 p、q 结点的指数域的值，进行相应操作，直到其中一个为空。具体操作如下。

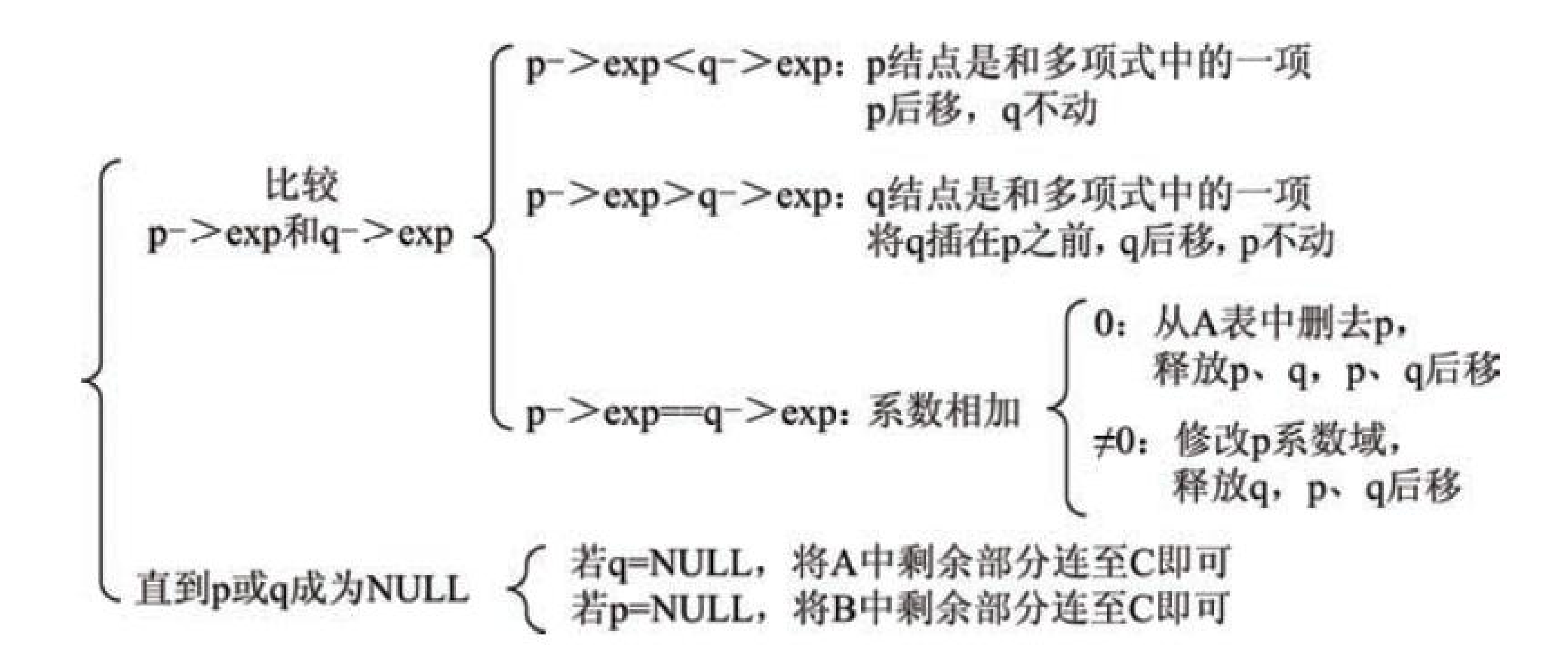


图 2.35 为多项式求和部分操作结果示意图，操作流程如下。

（1）r 指针后移一个结点指向 C 链表的第一个数据结点。

（2）r 结点的系数变为 2+6=8。

（3）p、q 指针各后移一个结点。

（4）比较 p、q 结点的指数域的值，将指数较大的 q 结点连入 C 链表。

（5）p 不动，q、r 各向前移动一步，此时链表状态如图 2.36 所示。

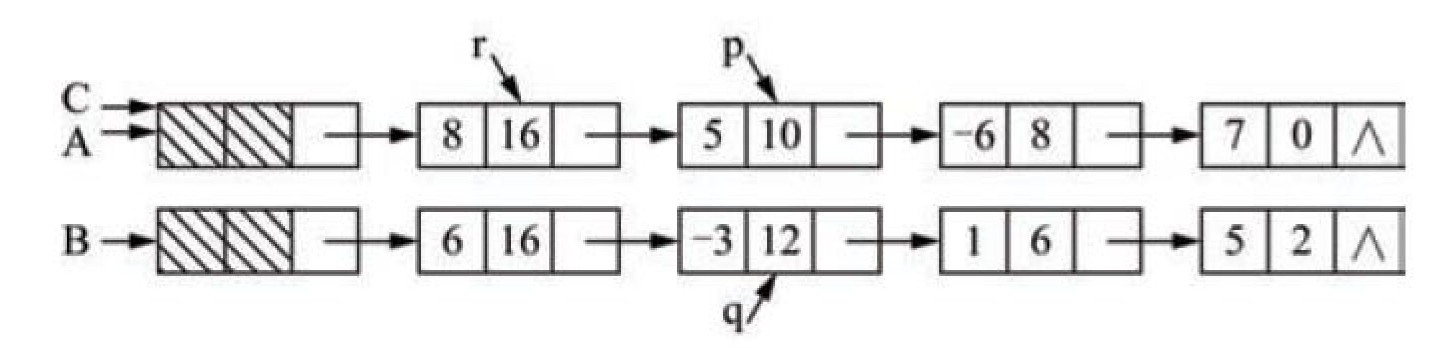


图 2.35　多项式求和第一步

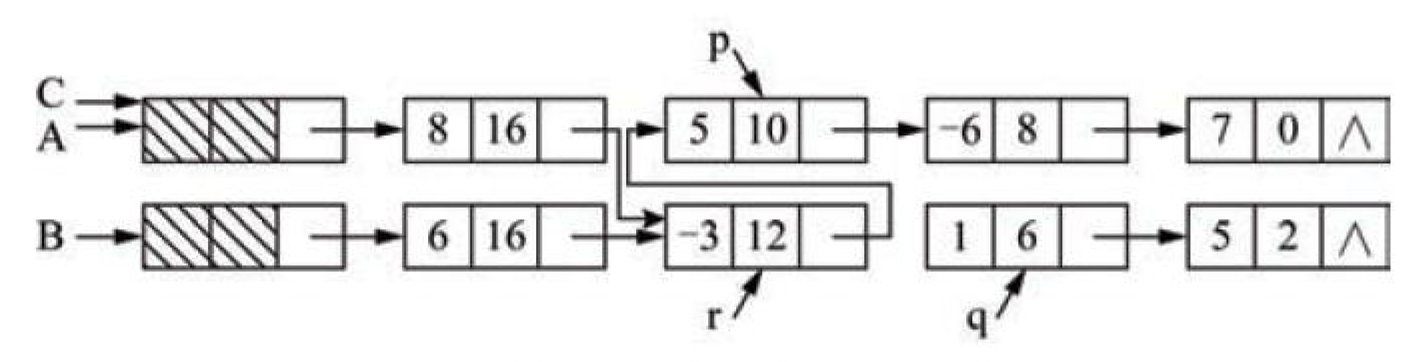


图 2.36　多项式求和第二步

算法伪代码如下：

```
Lpoly *add_poly(Lpoly *A, Lpoly *B) {
   p=A->next;
   q=B->next;
   r=A;
   C=A;
   while (p!=NULL) && (q!=NULL){ // 或者 while (!p && !q){
      if(p->exp==q->exp){
         x=p->coef+q->coef;
         if(x!=0) {
            p->coef=x;
            r=p;
         }else
            r->next=p->next;
         p=p->next;
         q=q->next;
      }else if(p->exp>q->exp) {
```

```
            r->next=p;
            r=p;
            p=p->next;
            } else {
                r->next=q;
                r=q;
                q=q->next;
            }
    }                                   // while 循环结环
    if(p==NULL)
            r->next=q;
    else
            r->next=p;
    return C;
}                                       // add_poly 函数结束
```

2.9 本章小结

本章为理解其他章节内容的基础，也是历年考查重点。学习过程应注意：

（1）重点理解线性结构的本质。

（2）顺序存储结构和链式存储结构的特征。

（3）熟练掌握特殊线性表——栈和队列的特殊性。

（4）熟练掌握串的性质、存储结构特点、常用算法实现及应用。

（5）掌握数组元素的存储方式，会计算数组元素的存储单元地址。

（6）掌握特殊矩阵的特征和存储方式。

（7）熟练掌握常见算法的原理和各种存储结构下的伪码实现。

第 3 章　树和二叉树

本章学习目标

- 理解树、二叉树的定义及它们的相关概念。
- 熟练掌握二叉树的基本性质、理解其证明过程。
- 掌握二叉树的两种存储结构。
- 熟练掌握二叉树的遍历。
- 熟练掌握二叉树的线索化及线索二叉树的特征。
- 熟练掌握 Huffman 树及其应用。
- 能够构造、应用二叉排序树、平衡二叉树。
- 掌握树、森林与二叉树之间的转化。
- 了解树和森林的遍历方法。

3.1　本章导学



树和二叉树

3.1.1　知识结构

本章知识结构如图 3.1 所示，加粗框中的内容需要考生重点理解并掌握。

3.1.2　命题特点

1．命题规律

（1）本章为各高校硕士研究生招生考试的重点考查内容，既有客观题又有主观题。本章知识点较多，考点相对丰富，出题形式灵活。

（2）本章可单独命题，也可与查找、排序等后续章节联合命题。

（3）树的相关概念、二叉树的性质、二叉树的遍历、线索二叉树、二叉排序树、平衡二叉树、树和森林的相关知识等易出客观题。

（4）二叉树的顺序存储结构易和堆排序等结合出主观题。

（5）二叉排序树、平衡二叉树易和查找联合出题。

2．命题趋势

本章在各高校硕士研究生入学考试中占据重要地位，其部分内容为后续章节的学习基础，主观、客观题目均易出。需要深刻理解基本概念、性质、存储结构特征、遍历的核心思想等，并在深入理解基本概念的基础上，熟练掌握其特征和应用。

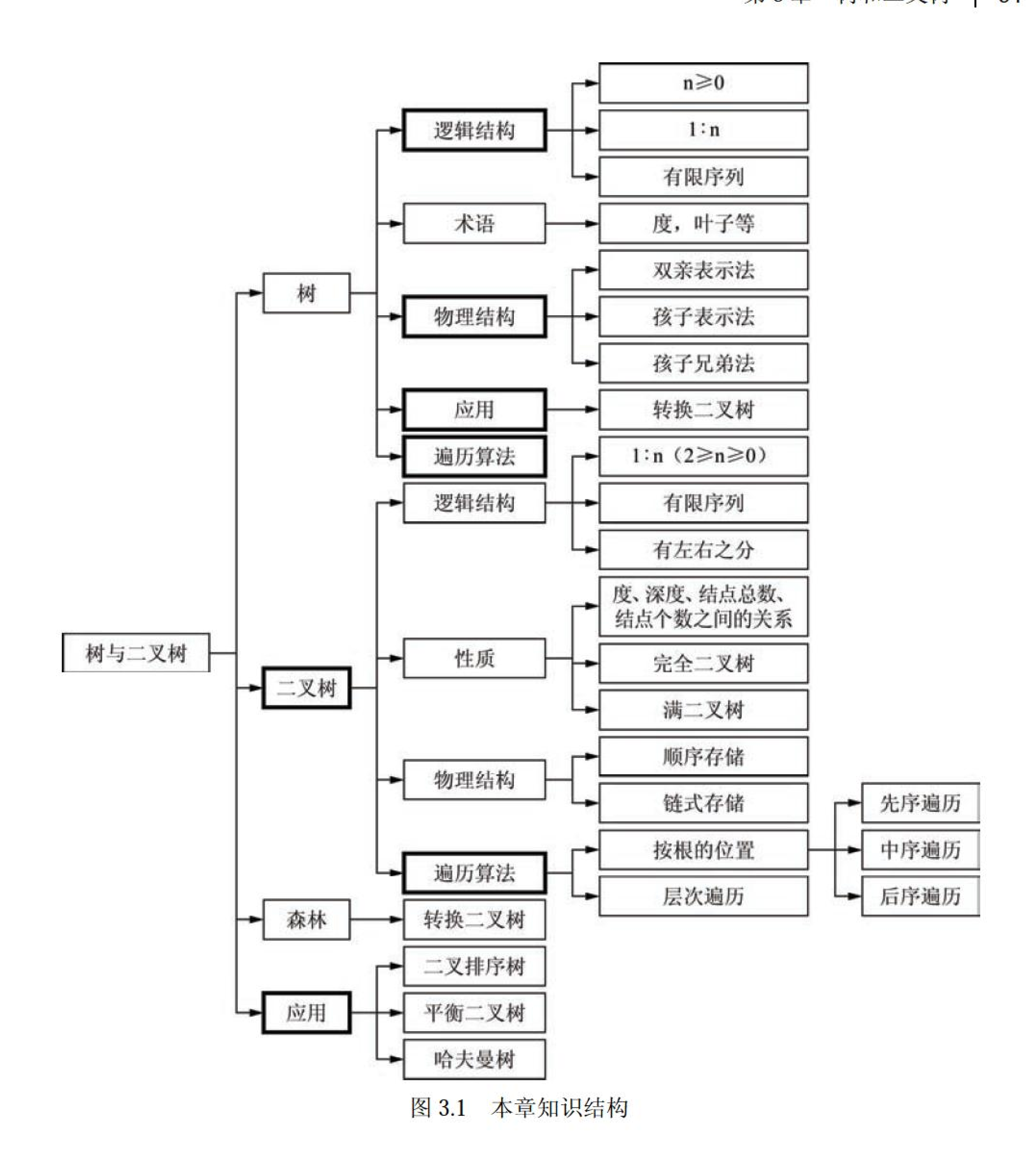


图 3.1　本章知识结构

3.2　树

树是一种常见的非线性数据结构，反映现实世界数据元素之间的 1∶n 逻辑关系，大多采用链式存储结构。为降低存储结构的复杂性，提出二叉树概念，其每个结点的孩子结点个数不超过 2 个，是应用最为广泛的树，一般将普通树形结构转换为二叉树进行处理。

3.2.1　定义

树是 n(n>=0)个结点的有限集。如果 n==0，称为空树；在任意一棵非空树中：

- 有且仅有一个特定的称为根的结点。
- 当 n>1 时，其余结点可分为 m（m>0）个互不相交的有限集合 $T_1,T_2,\cdots,T_m$，其中每个集合本身又是一棵树，并且称其为根的子树。
- 如果各子树的位置不能交换，称为有序树，否则，称为无序树。

显然，树是以递归形式定义的，即树的本质为递归。

图 3.2 为一棵根为 A 的树 $T=\{T_1,T_2,T_3\}$，图 3.3 为其一级子树，其中 $T_1=\{B,E,F,K,L,N\}$，$T_2=\{C,G\}$，$T_3=\{D,H,I,J,M,O,P\}$。

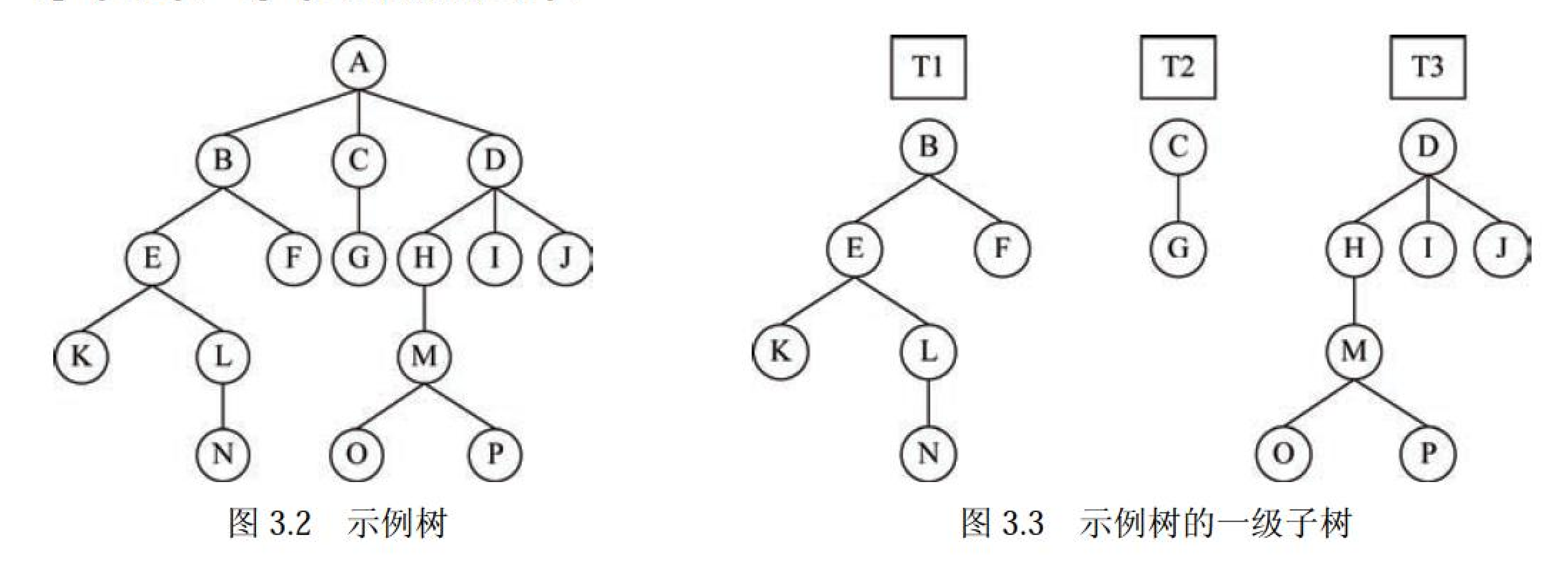


图 3.2　示例树　　　图 3.3　示例树的一级子树

3.2.2　树的表示形式

树的一般表示形式有树形表示法、文氏图表示法、凹入表示法、广义表表示法等，如图 3.4～图 3.7 所示。

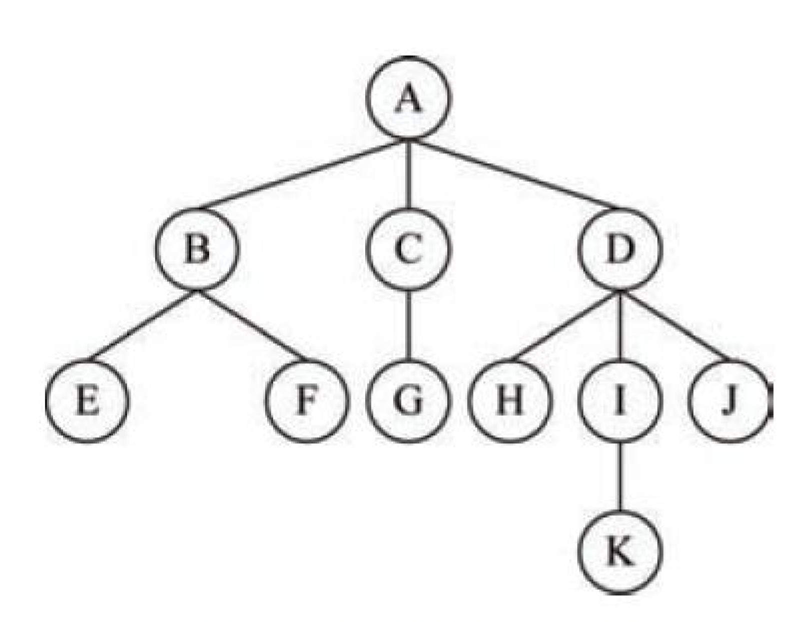


图 3.4　树的树形表示法

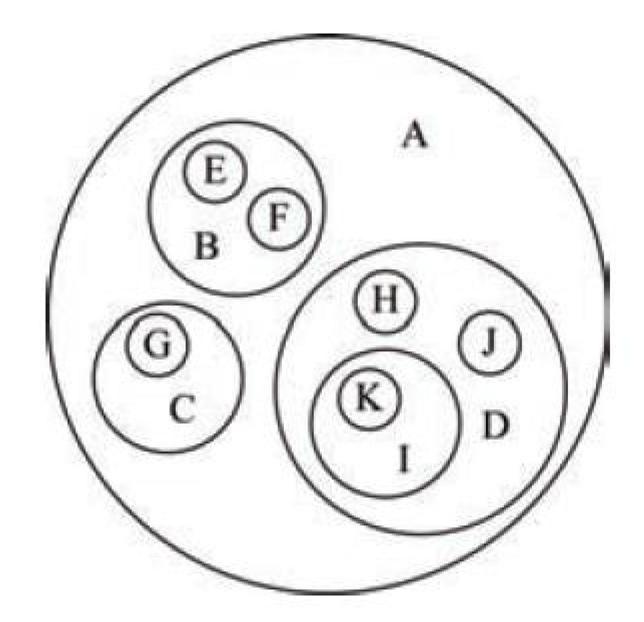


图 3.5　树的文氏图表示法

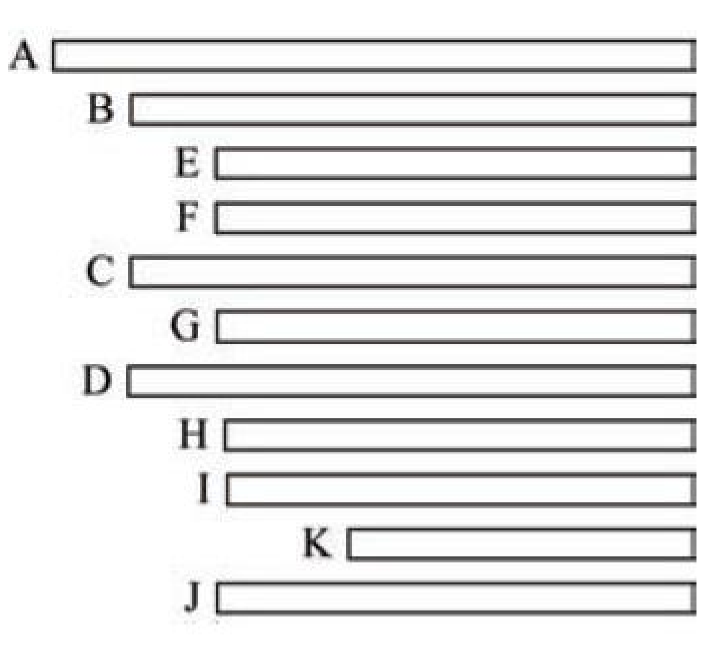


图 3.6　树的凹入表示法

（A（B（E，F），C（G），D（H，I（K），J））

图 3.7　树的广义表表示法

3.2.3　树的相关概念

树比较常用的相关概念如下。

（1）结点：包含一个数据元素及若干指向其所有子树的指针。

（2）结点的度：结点拥有的子树数。

（3）叶子（终端结点）：度为 0 的结点。

（4）分支结点（非终端结点）：度不为 0 的结点。

（5）树的度：树内各结点的度的最大值。

（6）孩子：某结点的子树的根结点称为该结点的孩子结点。

（7）双亲：某结点的直接前驱称为该结点的双亲结点。

（8）兄弟：同一双亲的子结点互为兄弟结点。

（9）祖先：从根结点到某结点所经分支上的所有结点均称为该结点的祖先结点。

（10）子孙：以某结点为根的子树中的任一结点都称为该结点的子孙。显然，双亲也可以是子孙，孩子也可以是祖先。

（11）层次：树的根为第 1 层，根的子结点为第 2 层，依此类推。如果考题上明确规定根所在的层次是 0，根的子结点就为第 1 层，考试中要灵活变通。

（12）堂兄弟：其双亲在同一层的结点互为堂兄弟。

（13）树的深度（高度）：树中结点的最大层次。

（14）无序树：树中任一结点的各子树之间无次序之分，即交换树中任一结点的各子树顺序不构成新树。一般情况下，树均定义为无序树。

（15）有序树：树中任一结点的各子树之间有次序之分，即交换树中任一结点的各子树顺序将构成新树。

（16）森林：m（m≥0）棵互不相交的树的集合。

3.2.4　树的抽象数据类型

```
ADT Tree{
    数据对象 D：D={ ai| ai|∈eleSet, i=1,2,…,n, n≥0}
    数据关系 R:
        若 D=∅，则 R=∅，称 tree 为空树。
        若 D≠∅，则 R={h}，h 为如下二元关系：
        (1) 在 D 中存在唯一的称为根的数据元素 root，该元素无前驱。
        (2) 若 D-{ root }≠∅，则存在 D-{ root }的一个划分{D1,D2,…，Dm}，m>0。其中：
            ①D1∩D2=∅，i≠j，i,j∈[1,m]。
            ②对于每一个 Di，存在唯一的 xi∈Di，使得<root, xi>∈h。
        (3) 对应于 D-{ root }的一个划分{D1,D2,…，Dm}，每个 Di 也是一棵树。
    基本操作 P:
        void initTree(*T);                  // 初始化一棵空树，返回根结点指针 T
        void creatTree(*T,n);               // 基于已存在的空树，创建含 n 个结
                                            // 点的树，返回根结点 T
        void clearTree(*T);                 // 清空一棵已存在的树，释放所有结点
```

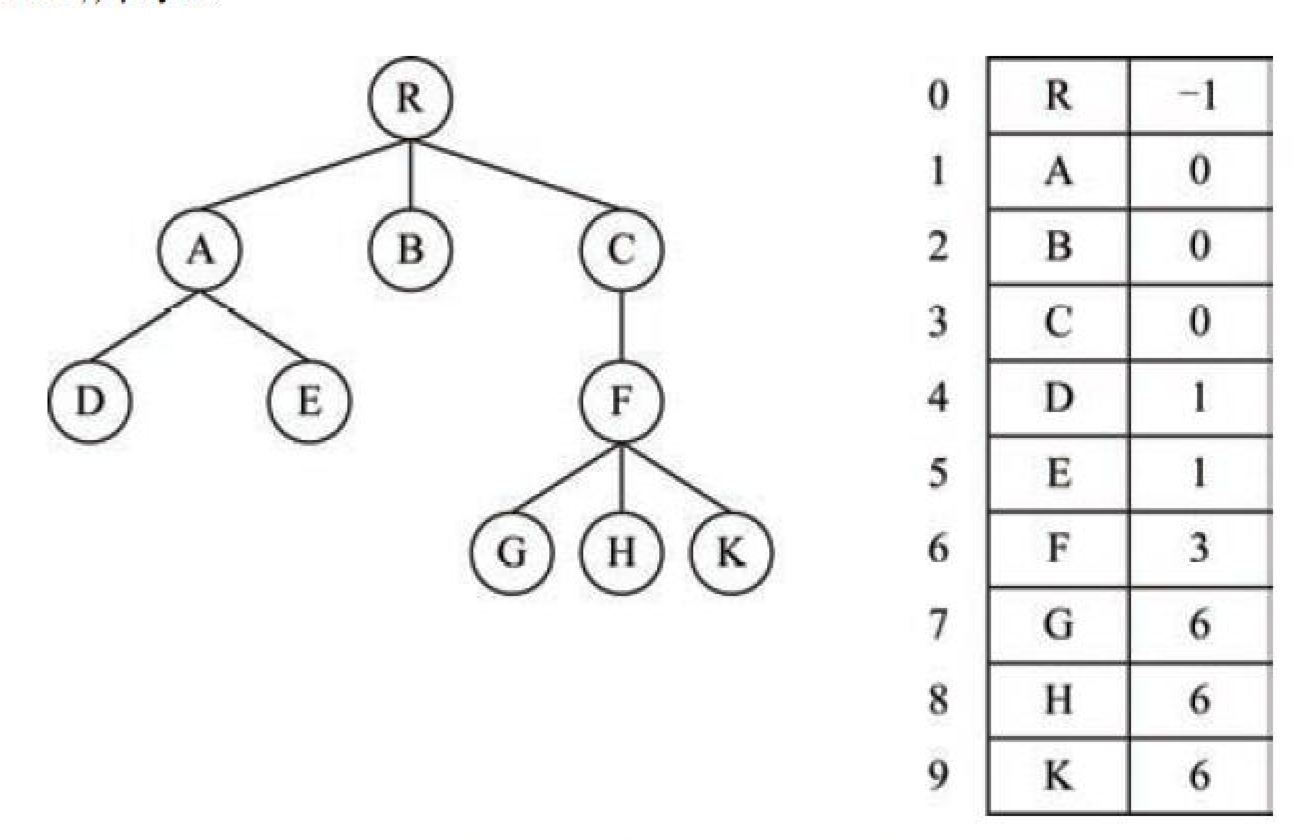
```
                                          // 空间，返回根结点 T
       int  emptyTree(*T);                // 判断树是否为空，空树返回 1，非空
                                          // 树返回 0
       unsigned  Depth(*T);               // 返回树 T 的深度
       treeNode treeRoot(*T);             // 返回树 T 的根结点
       void insertChild2Tree(*T,*p,i,c);  // 将c插入树T，作为p结点的第i棵子树
       void deleteChildFromTree(*T,*p,i); // 删除树 T 中 p 结点的第 i 棵子树
       void visitTree(*T,visit());        // 以 visit()方式遍历树 T
       void parent(*T,e);                 // 返回树 T 中结点 e 的双亲结点
} ADT Tree
```

3.2.5 存储结构

树的存储结构有以下三种表方法。

1. 双亲表示法

假设以一组连续空间存储树的结点，同时在每个结点中附设一个指针指示其双亲结点的位置，如图 3.8 所示。



0	R	−1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

图 3.8 树的双亲表示法

定义如下。

```
#define  maxTreeSize 1000
typedef  struct  pTreeNode{
    eleType data;                         // 结点值
    int parent;                           // 双亲位置
}parentTreeNode;
typedef  struct  pTree{
    pTreeNode  node[maxTreeSize];
   int  r;                                // 根的位置
   int  n;                                // 树的结点数
} parentTree;
```

2. 孩子结点表示法

树的孩子结点表示法有两种实现方式。

（1）多叉链表示：每个结点包含结点值和结点所有子树的孩子结点的指针。由于树

中的各个结点的度大小不一，导致结点中的指针域包含的指针个数不等，解决办法如下。

① 以树的度 d 为基准，每个结点均包含 d 个指针域。显然，这将浪费很多空间。

② 每个结点有几棵子树就包含几个指针域。由于各结点包含子树没有规定，操作难度较大。

（2）链式线性表：线性表的每个结点指向一个链表，假设树中结点的个数为 n，则树的链式线性表包含 n 个结点，每个结点包含两个域：

① 值域：存放该结点的值。

② 指向第一个子结点的指针域：即所有孩子结点链表的头指针。

孩子结点链表中的每个结点包含两个域：

① 值域：存放该结点的值。

② 指向下一个兄弟结点的指针域：指向同双亲的下一个兄弟结点。

树的孩子结点表示法如图 3.9 所示。

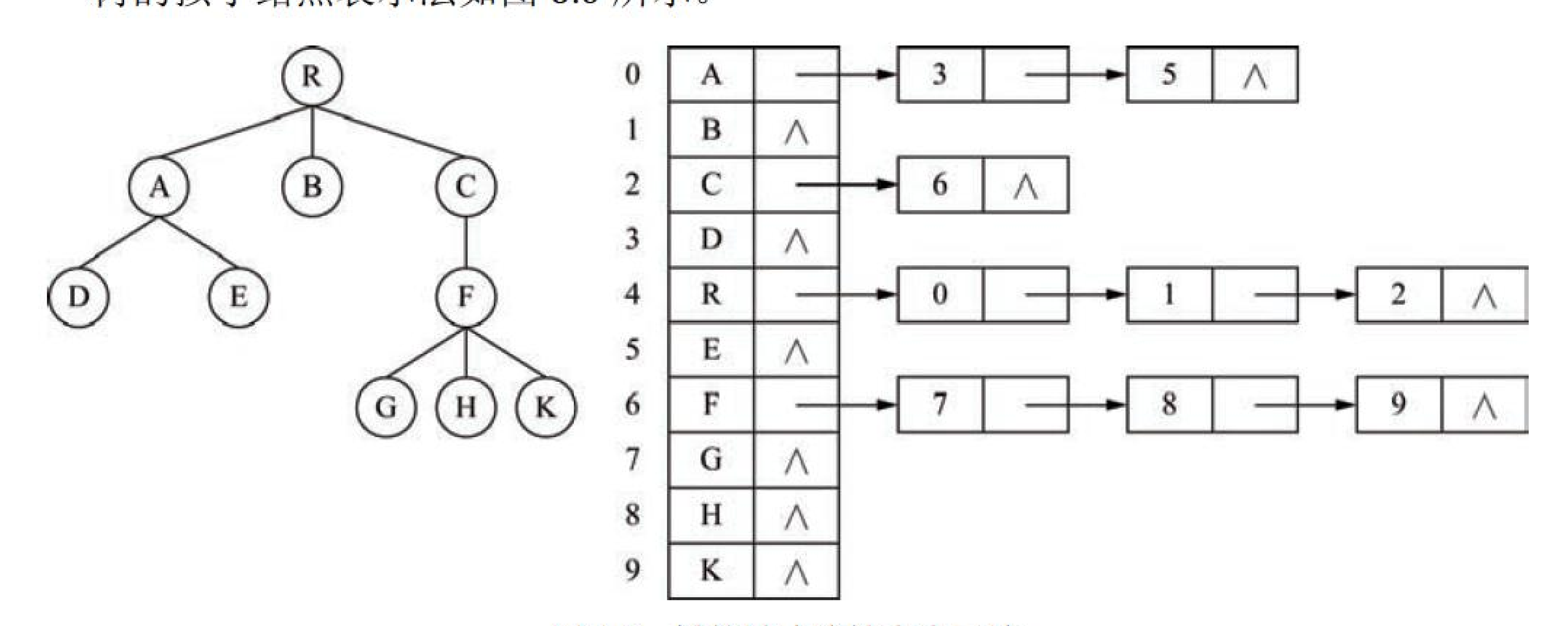


图 3.9　树的链式线性表表示法

定义如下。

```
#define  maxTreeSize 1000
typedef  struct  cTreeNode{
     int Child;
     struct cTreeNode  *next;                  // 下一个兄弟结点
} *ChildTreeNode;
typedef  struct  cTreeListNode{
     eleType data;
    ChildTreeNode  firstChild;                 // 孩子链表头指针
} ChildTreeListNode;
typedef  struct  cTree{
     ChildTreeListNode  node[maxTreeSize];     // 线性表
    int  r;                                    // 根的位置
    int  n;                                    // 树的结点数
}ChildTree;
```

3．孩子兄弟表示法

孩子兄弟表示法又称为二叉树表示法，或二叉链表表示法，即以二叉链表作树的存储结构。链表中的结点包含如下两个链域。

（1）孩子域：指向该结点的第一个孩子结点。

（2）兄弟域：指向该结点的下一个兄弟结点，如图 3.10 所示。

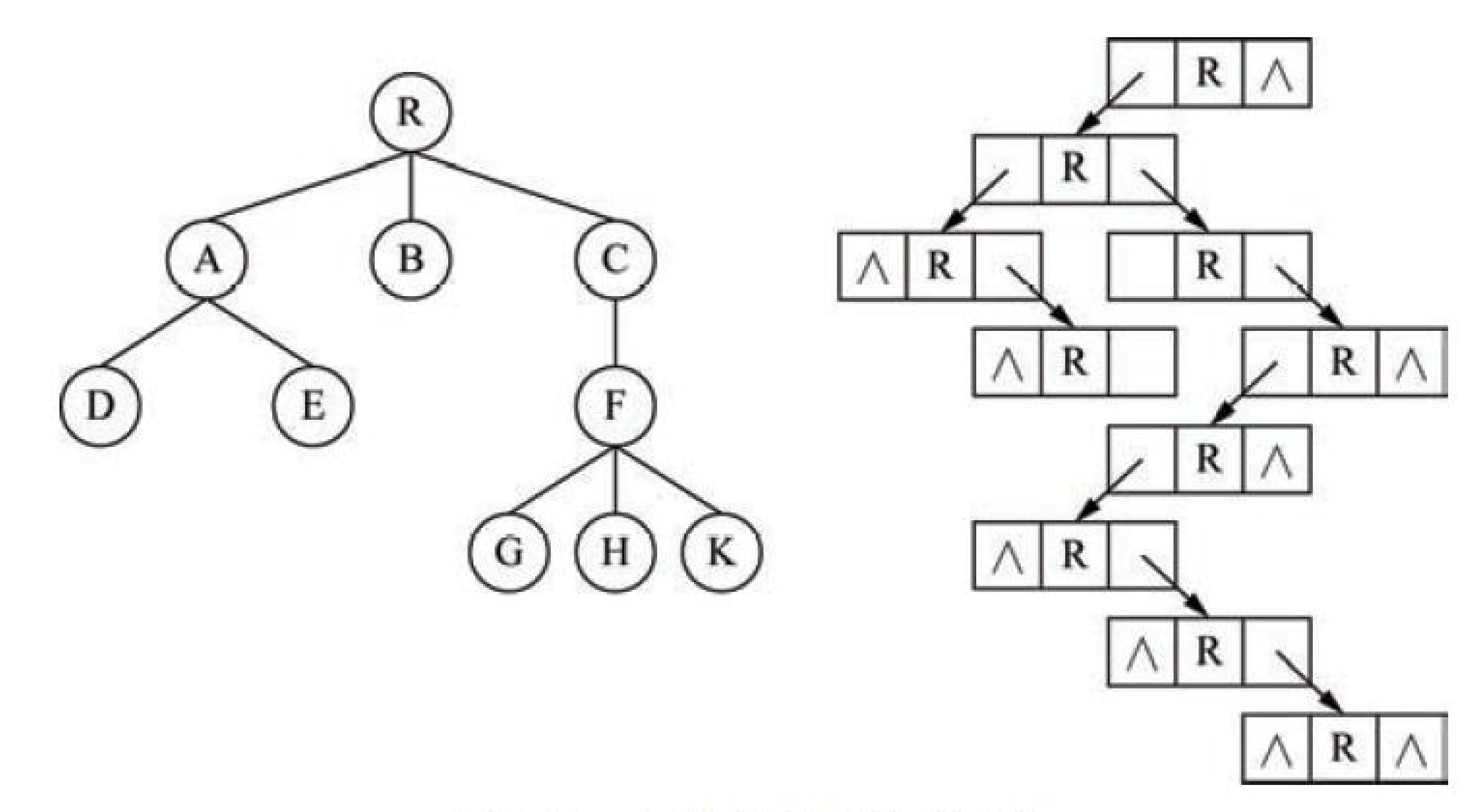


图 3.10　树的孩子兄弟表示法

定义如下：

```
Typedef struct csNode{
    ElemType data;
    struct csNode  *firstChild, *nextsibling;
} ChildSiblingNode, *ChildSiblingTree;
```

3.2.6　树的遍历

树的遍历指的是按照某种规则，对树中每一个结点访问一次。一般有 3 种遍历方法，分别为先序遍历、后序遍历和层次遍历。

1. 先序遍历

先序遍历首先访问树的根结点，然后从左至右逐一先序遍历根的每一棵子树。递归算法描述如下。

```
void preOrderTree (ChildSiblingTree root) {      // root 为树的根结点
    if(root!=NULL) {
           printf(root->data);                   // 访问根结点
             preOrderTree ( root->firstChild);
           preOrderTree ( root->nextsilbing);
     }
}
```

图 3.10 中的树的先序遍历序列为 RADEBCFGHK。

2. 后序遍历

后序遍历首先从左至右逐一后序遍历根的每一棵子树，最后访问树的根结点。递归算法描述如下。

```
void postOrderTree (ChildSiblingTree  root) {    // root 为树的根结点
    if(root!=NULL) {
```

```
            postOrderTree ( root->firstChild);
            printf(root->data);             // 访问根结点
            postOrderTree ( root->nextsilbing);
      }
}
```

图 3.10 中的树的后序遍历序列为 DEABGHKFCR。

3. 层次遍历

层次遍历按照如下顺序访问树中每一个结点：

（1）树的根结点入队列。

（2）队头结点出队，并访问该结点。

（3）将出队结点的所有孩子结点从左至右逐一入队。

（4）重复（2）和（3），直至所有结点均被访问过一次，算法描述如下。

```
#define  maxTreeNode  1000                          // 树结点最大个数
void levelOrderTree (ChildSiblingTree  root) {
    ChildSiblingNode  *q[maxTreeNode], p=root;     // 辅助队列
    int front=0, rear=0;                           // 队列置空
    printf("\n ");
    if(p!=NULL){                          // 也可以 if(p!=NULL) q[rear++]=p
    rear++;
    q[rear]=p;                            // 根结点进队
    }
while(front!=rear) {
    front++;
    p=q[front];                           // 队首结点出队
    printf(p->data);                      // 访问刚出队的结点
    if(p->firstChild!=NULL) {
            rear++;                       // P 为第一个孩子，不空则进队
            q[rear]=p->firstChild;
    }
    while(p->firstChild!=NULL) {          // 若 p 存在兄弟结点，则一一入队
            rear++;
            q[rear]=p->nextsilbing;
            p=p->nextsilbing;
        }
    }                                     // 当队列为空时，结束循环
}
```

图 3.10 中的树的层次遍历序列为 RABCDEFGHK。

3.3　二　叉　树

二叉树是一种特殊的树，由于其结构简单，一般树结构可以转换为二叉树，所以成为应用最广泛的树形结构。

二叉树可应用于通信及数据传送中的二进制编码、判定和决策、信息的检索和排序等。

3.3.1 定义

二叉树是特殊树形结构，是一个由 n 个结点组成的有限集合。这个集合或为空，或由一个称为根的结点以及两棵不相交的二叉树组成，这两棵二叉树分别称为根结点的左子树和右子树。其特点如下。

（1）二叉树可以为空，空二叉树不包含任何结点。

（2）每个结点至多包含两棵子树（即二叉树中不存在度大于 2 的结点，或者说二叉树的度不大于 2）。

（3）每个结点的两棵子树互不相交。

（4）二叉树的子树有左右之分，其次序不能颠倒。

有如下两种特殊的二叉树：

- 满二叉树：每层都达到最多结点数的二叉树。
- 完全二叉树：假设对满二叉树的结点进行连续编号，约定编号从根结点开始，自上而下，自左至右，依次递增。则深度为 k 有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 到 n 的结点一一对应时，称为完全二叉树。

图 3.11 为三个结点以内的二叉树形态。其中图 3.11（a）为空二叉树，图 3.11（b）为只有一个结点的二叉树，图 3.11（c）和图 3.11（d）为含 2 个结点的二叉树的所有形态，图 3.11（e）～图 3.11（i）为包含 3 个节点的二叉树的所有形态。

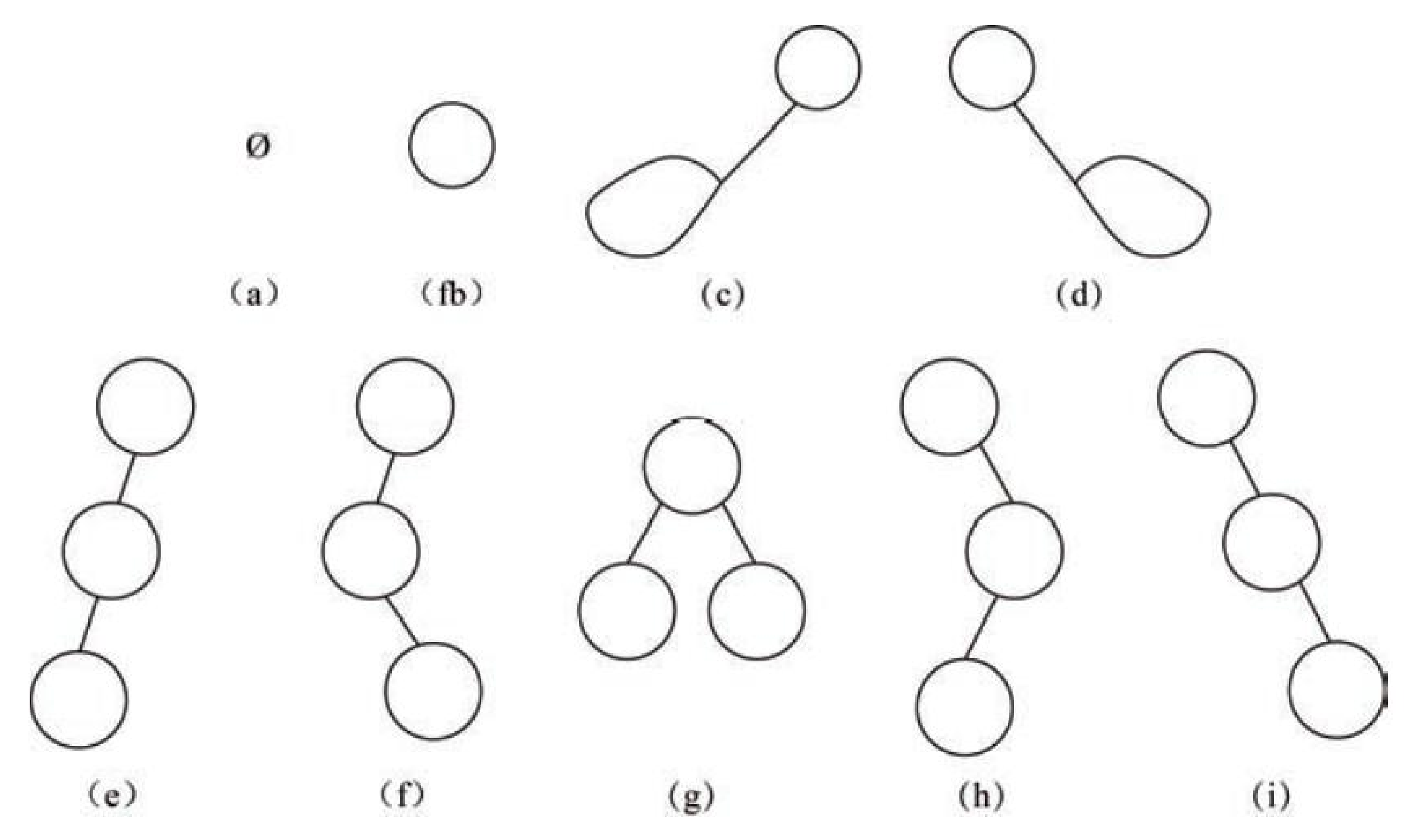


图 3.11　三个结点以内的二叉树形态

二叉树的抽象数据类型描述如下。

```
ADT  BiTree {
   数据对象 D：D 是具有相同特性的数据元素的集合。
   数据关系 R：
      若 D=Ø，则 R=Ø；                           // 空二叉树
```

```
    若 D≠Ø，则 R={H}，H 为如下二元关系：
    (1) D 中存在唯一称为根的元素 root，该元素无前驱        // 唯一无前驱结点：根
    (2) 若 D-{root}≠Ø，则存在 D-{root}={D_L,D_R}，D_L∩D_R= Ø   //子树之间无交集
        ① 若 D_L≠Ø，则存在 x_L∈D_L（x_L 唯一），<root, x_L>∈H，且存在 D_R 上的关系 H_R
    ∈H，H={< root, x_L>,< root, x_R>,H_L,H_R}。
        ②(D_L,{ H_L })是一棵二叉树，称为 root 的左子树。
        ③(D_R,{ H_R })是一棵二叉树，称为 root 的右子树。
  基本操作 P:
    initBiTree( &T );
    destroyBiTree(&T);
    createBiTree(&T);
    clearBiTree(&T);
    BiTreeEmpty(T);
    BiTreeDepth(T);
    rootBiTree(T);
    valueBiTree(T,e);
    assignBiTree(T,&e,value);
    parentBiTree(T,e);
    leftChildBiTree(T,e);
    rightChildBiTree (T,e);
    leftSiblingBiTree(T,e);
    rightSiblingBiTree(T,e);
    insertChildBiTree(T,p,LR,c);
    deleteChildBiTree(T,p,i);
    preOrderBiTree( T,visit() );
    inOrderBiTree( T,visit() );
    postOrderBiTree( T,visit() );
    levelOrderBiTree( T,visit() );
}ADT BiTree
```

3.3.2　性质

性质 1：任何二叉树的第 i 层最多有 2^{i-1} 个结点（i≥1）。

证明：数学归纳法

（1）二叉树的第 1 层只有一个结点。所以，当 i=1 时，$2^{i-1}=2^0=1$ 成立。

（2）假设结论对第 i 层成立，即第 i 层最多有 2^{i-1} 个结点。

（3）由于二叉树每个结点的度最多为 2，因此第 i+1 层结点的个数，最多应该是第 i 层结点个数的 2 倍，即 $2\times2^{i-1} = 2^i$。

命题得证。

性质 2：深度为 k 的二叉树最多有 2^k-1 个结点（k≥1）。

证明：

（1）由性质 1 知，二叉树的第 i 层最多有 2^{i-1} 个结点。

（2）假设深度为 k 的二叉树最多结点数如下：

$$2^0+2^1+\cdots+2^{k-1} = 2^k-1$$

命题得证。

性质 3：对任何一棵二叉树，如果其终端结点（度为 0）的个数为 n0，度为 2 的结点个数为 n2，则 n0 = n2 + 1。

证明：

（1）设二叉树中度为 1 的结点个数为 n1，则二叉树总的结点个数 n=n0+n1+n2。

（2）二叉树中除根结点外，其余每个结点都有一个向上的分支指向其双亲结点。假设二叉树的总边数为 m，则二叉树的总边数 m 和结点个数 n 的关系为：m= n −1。

（3）二叉树的总边数由度为 1 的结点所附边数和度为 2 的结点所附边数构成，度为 1 的结点附 1 条边，度为 2 的结点附 2 条边。所以，总边数 m 和度为 1 的结点数 n1，以及度为 2 的结点数 n2 的关系为： m=1*n1+2*n2。

（4）由（1）～（3）可知：n0 = n2 + 1。

命题得证。

性质 4：具有 n 个结点的完全二叉树的深度为 $\log_2 n+1$。

证明：由性质 2 反推可以直接得到结论。

性质 5：如果对一棵有 n 个结点的完全二叉树的结点按层次编号（从第一层到第 $\log_2 n+1$ 层，每层从左至右），则对任一结点 i 有：

- 如果 i=1，则结点 i 是二叉树的根结点，无双亲。
- 如果 i>1，则其双亲 parent(i)为结点 $\lfloor i/2 \rfloor$。
- 如果 2i>n，则结点 i 无左孩子；否则其左孩子 lChild(i)为第 2i 个结点。
- 如果 2i + 1>n，则结点 i 无右孩子；否则其右孩子 rChild(i)为第 2i+1 个结点。

性质 6：深度为 k 且包含 2^{k-1} 个结点的二叉树称为满二叉树。对满二叉树的结点可以从根结点开始自上向下、自左至右顺序编号，图 3.12 为深度为 4 的满二叉树，其中每个结点圈中的数字即为该结点的编号。

性质 7：深度为 k，含有 n 个结点的二叉树，当且仅当其每个结点的编号与相应满二叉树结点的编号从 1 到 n 相对应时，称此二叉树为完全二叉树，如图 3.13 所示。

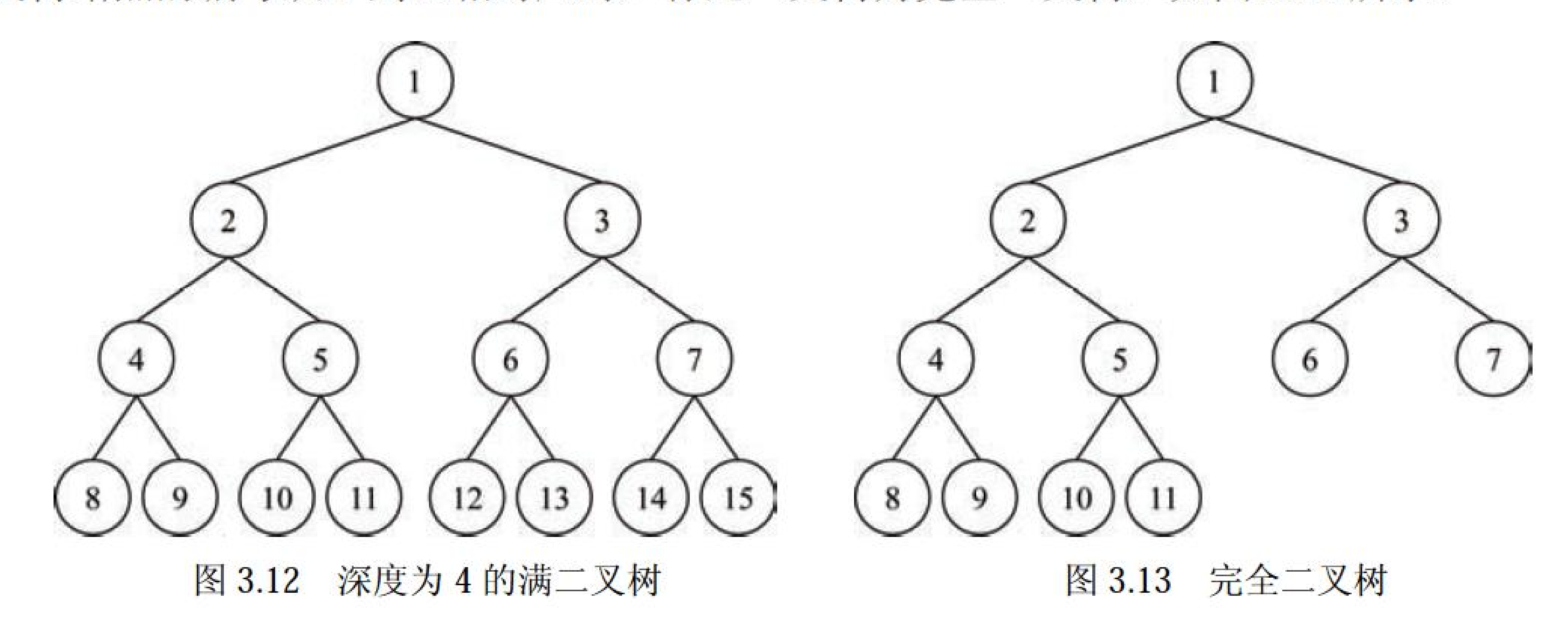


图 3.12 深度为 4 的满二叉树　　图 3.13 完全二叉树

3.3.3 存储结构

1．顺序存储结构

自上而下、自左至右将完全二叉树上的结点存储到一维数组中，即将完全二叉树上

编号为 i 的结点存储在一维数组中下标为 i－1 的分量中，如图 3.14 所示。对于一般二叉树，则应将其每个结点与完全二叉树上的结点位置相对应，存储到一维数组的相应元素中，如图 3.15 所示。

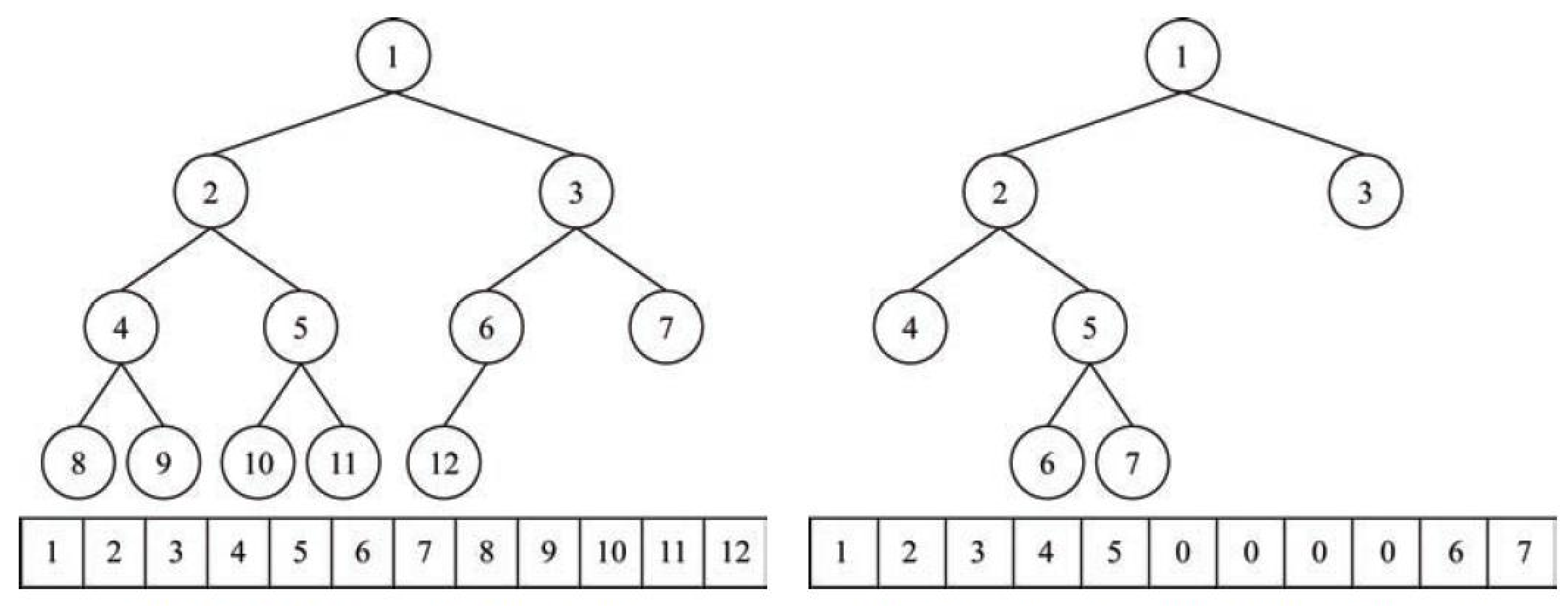


图 3.14　完全二叉树的顺序存储示　　图 3.15　非完全二叉树的顺序存储示意图

二叉树的顺序存储结构定义如下：

```
#define maxTreeSize 1000
typedef treeEleType SqBiTree[maxTreeSize];
SqBiTree bt;
```

其中，bt 是一维数组，每个数组元素存储树的一个结点的数据信息。假定 0 号位置（即 bt[0]）空置不用，从 1 位置开始存储二叉树的所有元素。按照完全二叉树结点的编号自上而下、从左至右的顺序将每个元素存入数组中。

对于一般二叉树，顺序存储结构仍然需要同深度完全二叉树所含结点数的元素个数，比较浪费空间。比如，由图 3.16 含 3 个结点的单边二叉树及其顺序存储结构可以看出：

（1）二叉树仅包含 3 个结点，但顺序存储空间占 7 个数组元素空间。

（2）深度为 k 的二叉树至少包含 k 个结点（如图 3.12 的单边二叉树）。

（3）采用顺序存储结构，深度为 k 的二叉树至少需要 2^{k-1} 个数组元素的存储空间（第 k 层只有最左边的一个结点），至多需要 2^k-1 个数组元素的存储空间（存在第 k 层最右边的那个结点）。

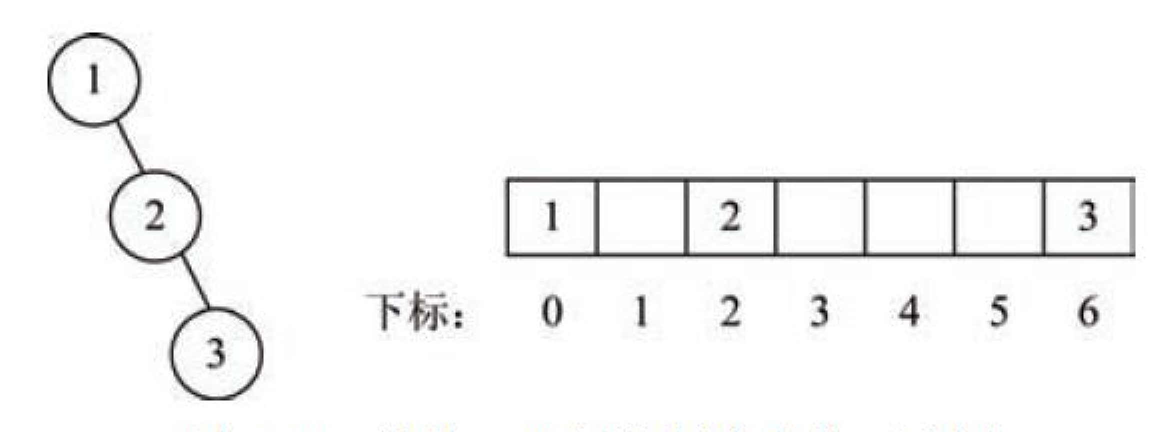


图 3.16　单边二叉树的顺序存储示意图

由（2）、（3）可知，采用顺序存储结构时，深度为 k 的二叉树至少需要的数组元

素个数和至少包含的结点个数之间的差为 $2^{k-1}-1$，也就是可能浪费 $2^{k-1}-1$ 个数组元素的存储空间。

所以，顺序存储结构适用于满二叉树和完全二叉树，但并不适用于一般二叉树。

2．链式存储结构

由于存在大量的空间浪费或对总体空间需求不可知，非完全二叉树和动态变化的二叉树均不适合采用顺序存储结构。二叉树的链式存储结构可以适用这种需求，其中每个结点需要包含结点本身的数据信息，以及指向其子树或双亲的分支信息。据此，二叉树的链式存储结构主要有二叉链表和三叉链表，如图 3.17 所示。

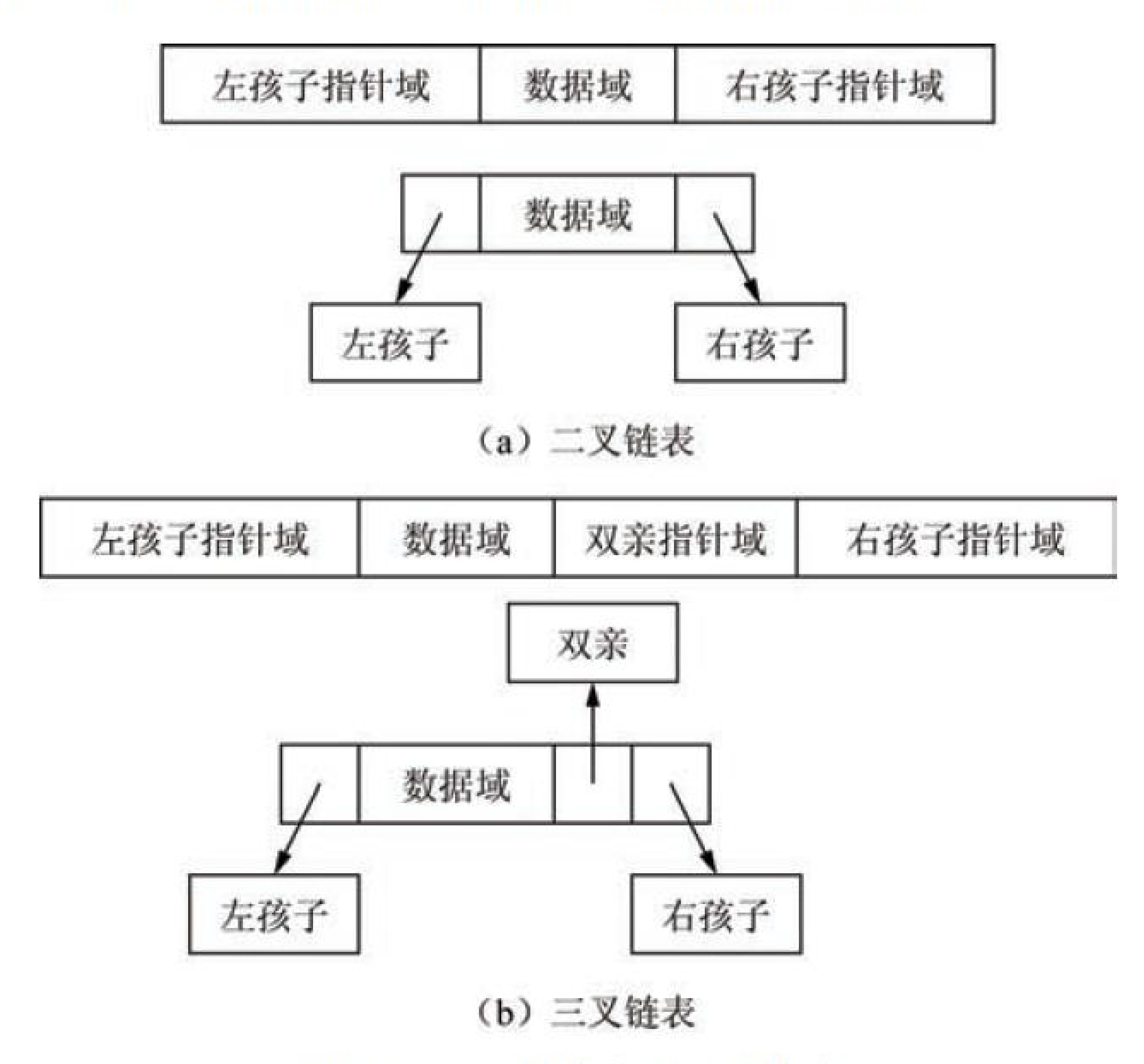


图 3.17　二叉链表和三叉链表

（1）二叉链表：二叉树的每个结点包含 3 个域，分别为结点本身的信息、左子树根结点存储位置以及右子树根结点存储位置，如图 3.18 所示。该存储结构有利于求某结点的左右孩子结点，但求某结点的双亲结点不太方便。

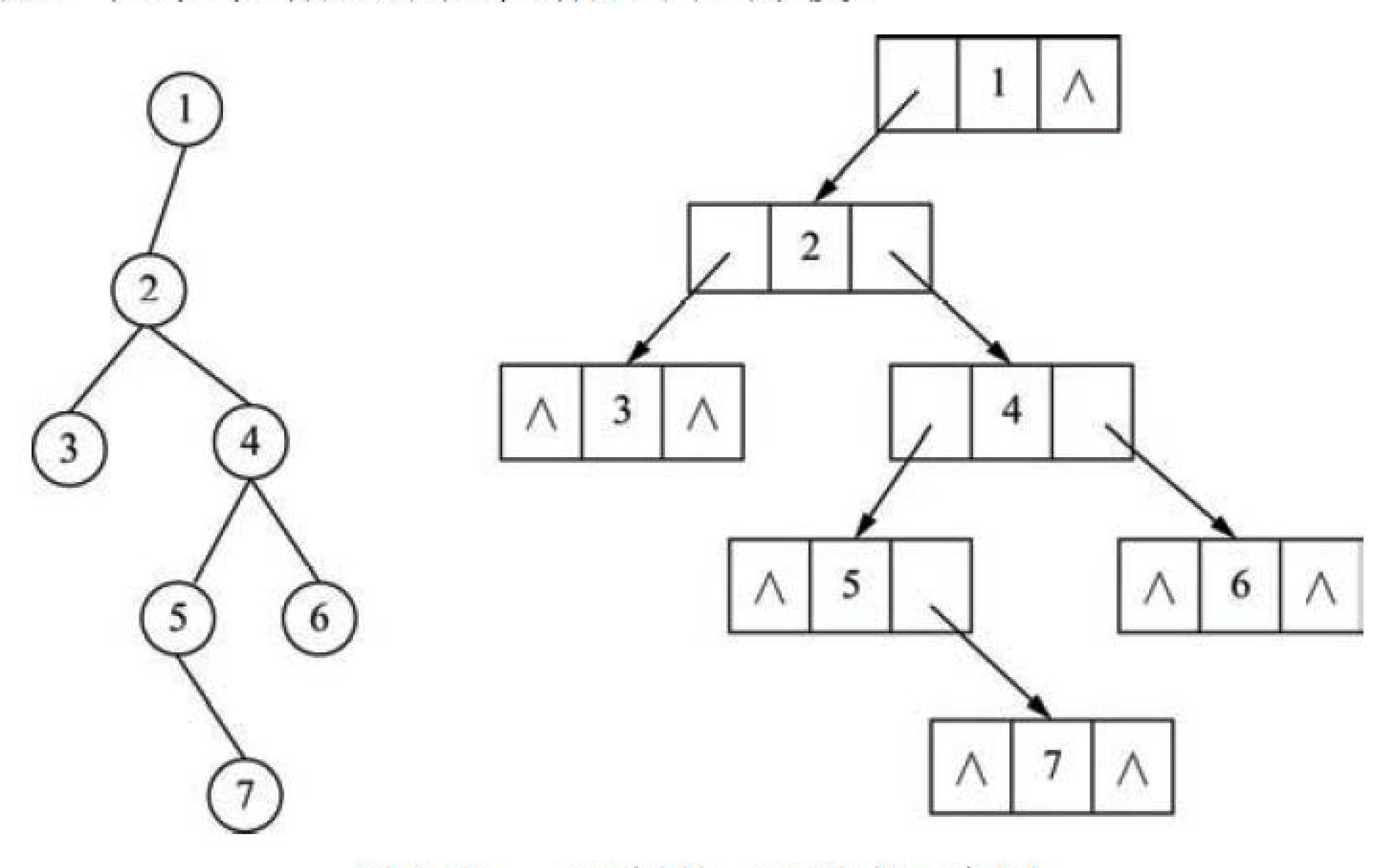


图 3.18　二叉树的二叉链表示意图

二叉树的二叉链表存储方式定义如下。

```
typedef struct BiNode{
    eleType data;
    struct BiTNode  *lChild, *rChild;
}BiNode, * BiTree;
```

（2）三叉链表：二叉树的每个结点包含 4 个域，分别为结点本身的信息、双亲结点存储位置、左子树根结点存储位置以及右子树根结点存储位置，如图 3.19 所示。该存储结构对于求某结点的左右孩子结点以及双亲结点都很方便。

二叉树的三叉链表存储方式定义如下。

```
typedef struct TriTNode{
    TelemType data;
    struct TriTNode  *parent, *lChild, *rChild;
}TriTNode, * TriTree;
```

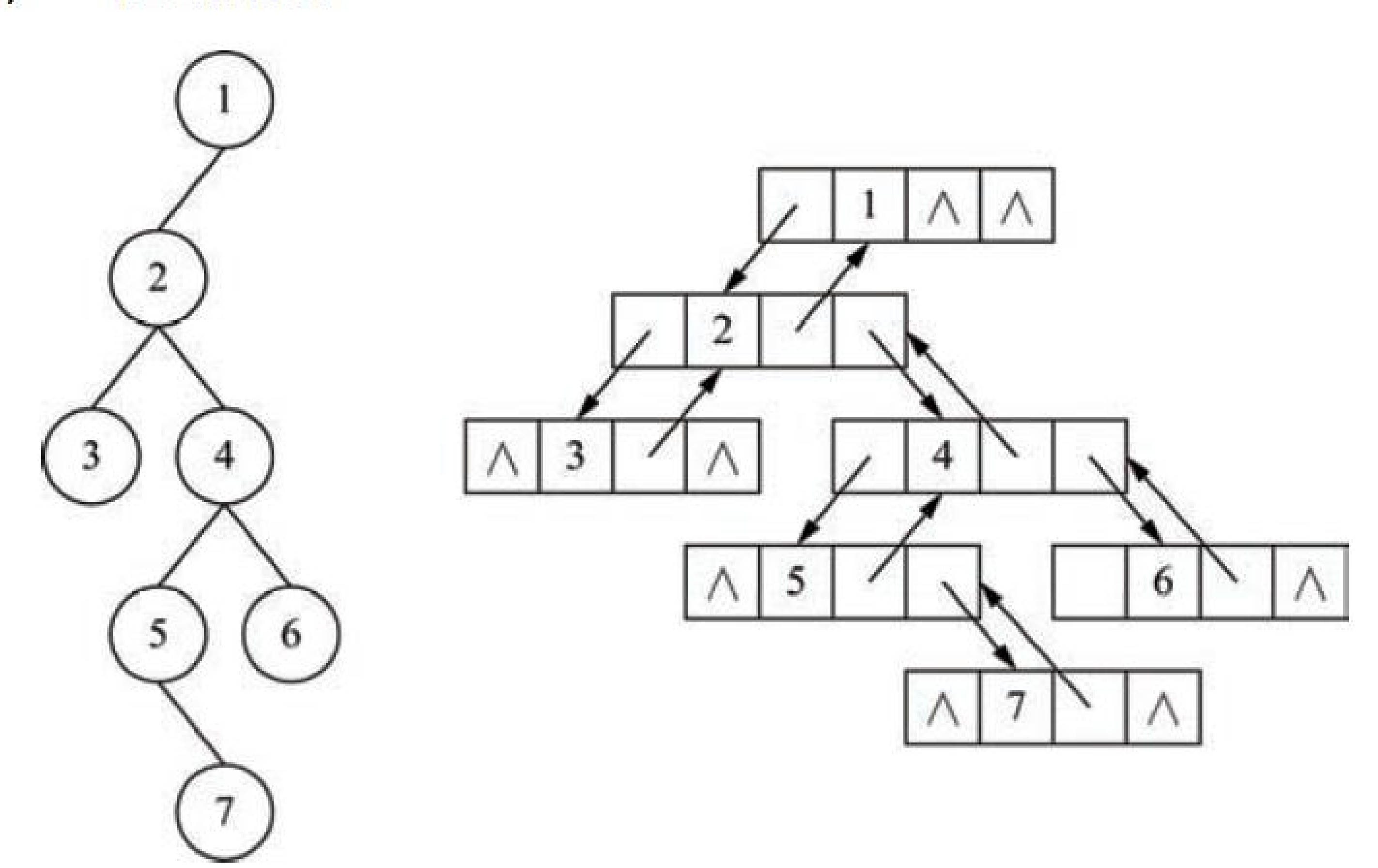


图 3.19　二叉树的三叉链表示意图

3.3.4　二叉树的遍历

1. 概念

遍历是指按一定次序将数据集中的每个数据元素访问一遍，且每个数据元素只能访问一次。访问是指对结点进行的各种操作，比如输出、查找、修改等。由于线性结构比较简单，元素之间只存在前后序关系，所以没有提出遍历的概念。而非线性结构元素之间的关系比较复杂，需要规定某种策略对所有元素进行访问。

二叉树的遍历是指以一定的次序访问二叉树中的每个结点，且每个结点仅被访问一次。假设遍历二叉树时访问结点的操作是输出结点数据域的值，则对二叉树的遍历结果将得到一个线性序列。所以，遍历二叉树的实质是把二叉树的结点进行线性排列，这是进行二叉树其他操作的基础。

二叉树每个数据元素对应的结点由三部分组成：根结点（T）、左子树（L）、右子树（R）。若能依次遍历这三个部分，也就遍历了整个二叉树。按照一般先左后右的习惯，

对这三部分的访问对应三种遍历方法，均为递归定义。

（1）先序遍历：根—左子树—右子树（TLR）。

首先访问根结点；然后先序遍历根结点的左子树；最后先序遍历根结点的右子树。

（2）中序遍历：左子树—根—右子树（LTR）。

首先中序遍历根结点的左子树；然后访问根结点；最后中序遍历根结点的右子树。

（3）后序遍历：左子树—右子树—根（LRT）。

首先后序遍历根结点的左子树；然后访问根结点；最后后序遍历根结点的右子树。

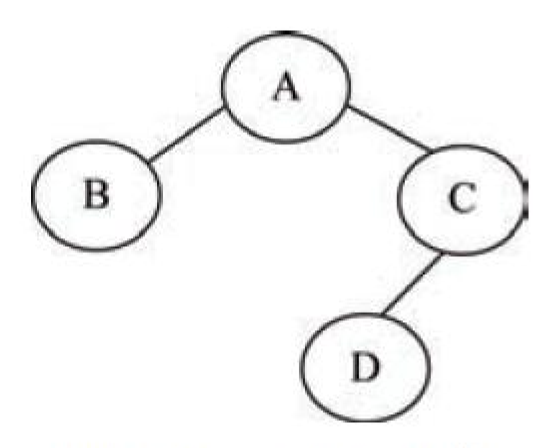


图 3.20　示例二叉树

三种遍历具体如下，并以图 3.20 所示二叉树为例。

1. 先序遍历

图 3.20 二叉树的先序遍历流程如图 3.21 所示，遍历序列为 ABCD。

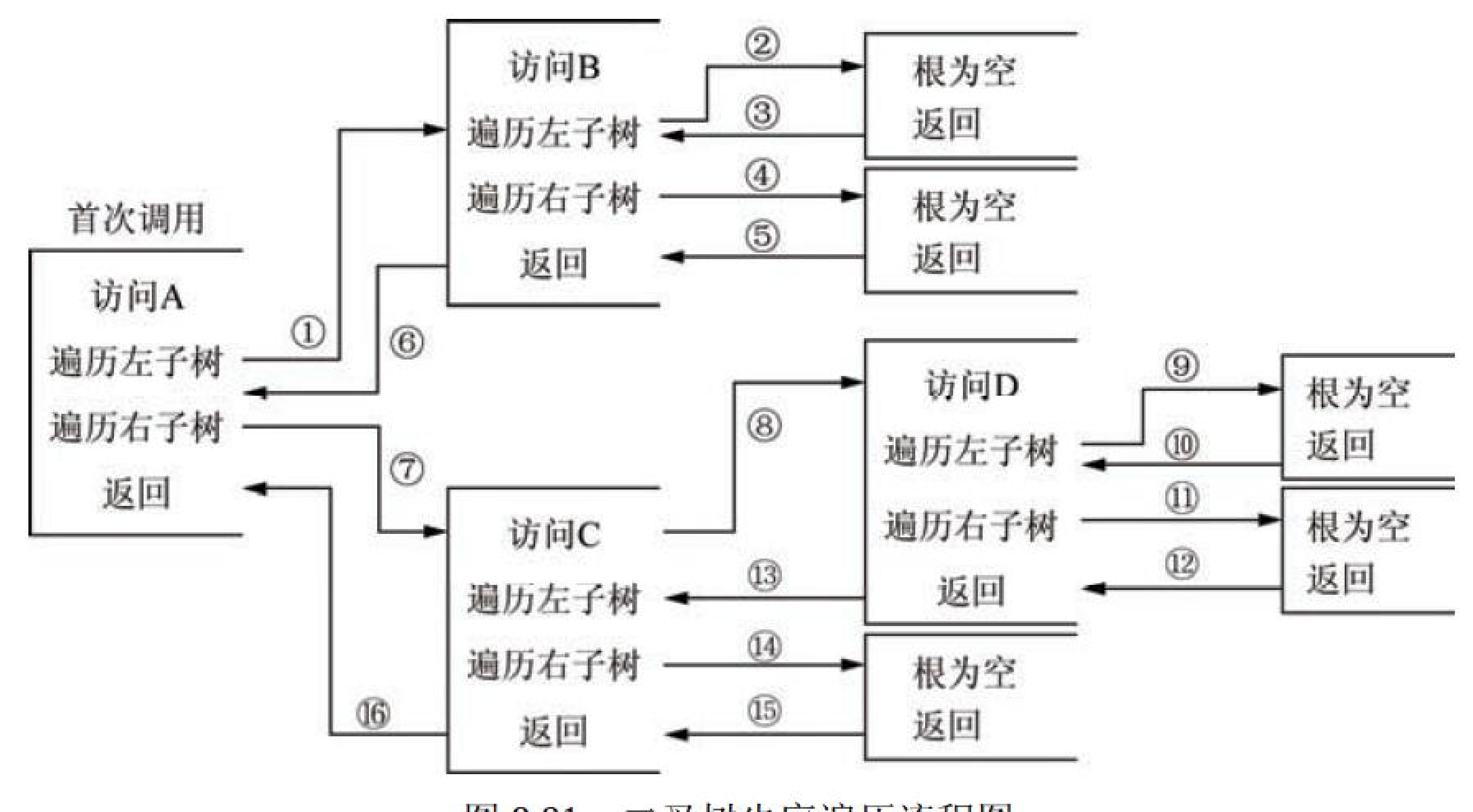


图 3.21　二叉树先序遍历流程图

1）递归算法

二叉树的二叉链表存储结构先序遍历递归算法伪代码描述如下。

```
void preOrderBiTree (BiTree T , visit ()) {
     if(T) {
       visit(T->data);
       preOrderBiTree (T->lChild, visit);
       preOrderBiTree (T->rChild,visit);
    }
}
```

2）非递归遍历算法

递归算法简明精练，易于理解，但系统需要维护一个工作栈以保证递归函数的正确执行，故效率低下；而且在某些高级语言中，不提供递归调用的语句及功能。为提高程序设计的能力，有必要研究非递归算法及其实现。

非递归算法的关键问题在于如何实现由系统完成的递归工作栈。实际应用中可通过

人为设置的栈模拟系统工作栈，完成递归算法的非递归化。

先序遍历的递归算法分析如下。

每进行一次深层次调用都需要保留当前调用层上的一些信息，主要是指向当前层根结点的指针。

在先序遍历的非递归算法中，设置一个栈 S 来存放所经过的根结点的指针，方便在访问完某个结点及其左子树后，继续访问此结点的右子树.

所以，递归算法的非递归化需要在访问完某结点后，将该结点的指针保存在栈中。

先序遍历的非递归过程如下。

（1）设置一个空栈。

（2）若二叉树不空，访问根结点，并将其指针入栈。

（3）按（2）中规则遍历其左子树。

（4）左子树遍历结束后，若栈不空，将栈顶元素出栈，按（2）中规则遍历栈顶元素的右子树。

（5）直至栈空为止。

二叉树的二叉链表存储结构先序遍历非递归算法伪代码描述如下。

```
void fpreOrderBiTree (BiTree T) {
    int top=-1;       // 利用数组实现堆栈，仅在 top 下标处增减元素，top=-1 为空栈
    while(T!=NULL||top!=-1) {
        while(T!=NULL) {
            printf(T->data);
            top++;
            s[top]=T;
            T=T->lChild;
}
if(top!=-1) {
            T=s[top];
            top--;
            T=T->rlChild;
        }
    }
}
```

2. 中序遍历

图 3.20 中二叉树的中序遍历流程如图 3.22 所示，对先序遍历的顺序进行如下调整。

（1）第 ①～③ 步为递归调用过程。

（2）第一个输出结点为第 ④ 步的 B。

（3）第 ⑤～⑦ 步为递归调用和返回。

（4）第二个输出结点为第 ⑧ 步的 A。

（5）第 ⑨～⑫ 步为递归调用和返回。

（6）第三个输出结点为第 ⑬ 步的 D。

（7）第 ⑭～⑯ 步为递归调用和返回。

（8）第四个输出结点为第 ⑰ 步的 C。

（9）第⑱～⑳步为递归调用和返回。

此时已返回至最顶层调用，中序遍历结束，遍历序列为 BADC。

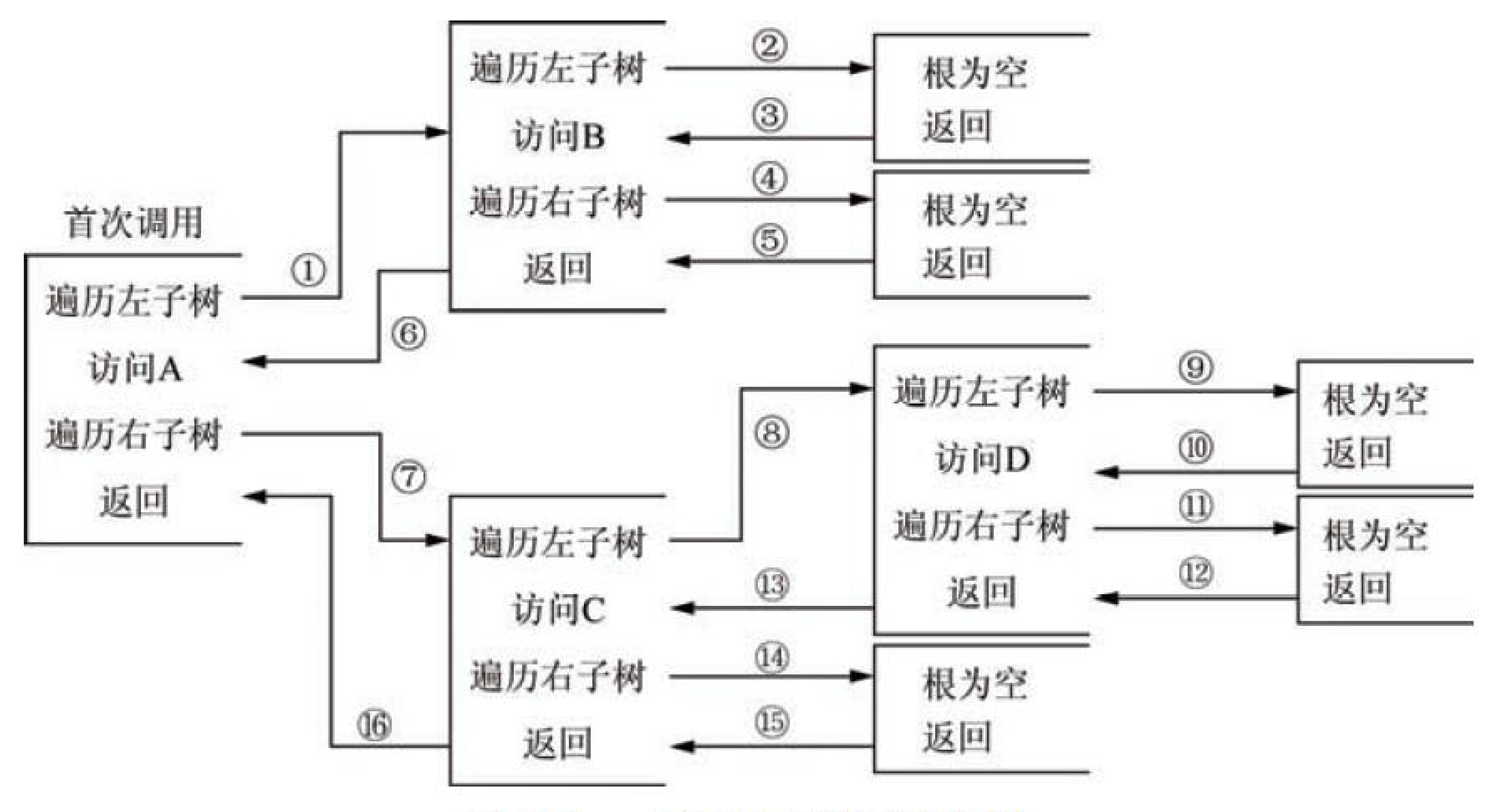


图 3.22　二叉树中序遍历流程图

1）递归算法

二叉树的二叉链表存储结构中序遍历递归算法伪代码描述如下。

```
void inOrderBiTree (BiTree T , visit ()) {
     if(T) {
       inOrderBiTree (T->lChild, visit);
       visit(T->data);
       inOrderBiTree (T->rChild,visit);
    }
}
```

2）非递归遍历算法

二叉树的二叉链表存储结构中序遍历非递归算法伪代码描述如下。

```
void finOrderBiTree (BiTree T) {
    int top=-1;       // 利用数组实现堆栈，仅在 top 下标处增减元素，top=-1 为空栈
    while(T!=NULL||top!=-1) {
        while(T!=NULL) {
            s[++top]=T;
            T=T->lChild;
        }
        if(top!=-1) {
            T=s[top--];
            printf(T->data);
            T=T->rChild;
        }
    }

}
```

3. 后序遍历

图 3.20 中二叉树的后序遍历流程如图 3.23 所示，对先序遍历的顺序进行如下调整。

（1）第 ①～⑤ 步为递归调用过程。
（2）第一个输出结点为第 ⑥ 步的 B。
（3）第 ⑦～⑬ 步为递归调用和返回。
（4）第二个输出结点为第 ⑭ 步的 D。
（5）第 ⑮～⑰ 步为递归调用和返回。
（6）第三个输出结点为第 ⑱ 步的 C。
（7）第 ⑲ 步为递归返回。
（8）第四个输出结点为第 ⑳ 步的 A。

此时已返回至最顶层调用，后序遍历结束，遍历序列为 BDCA。

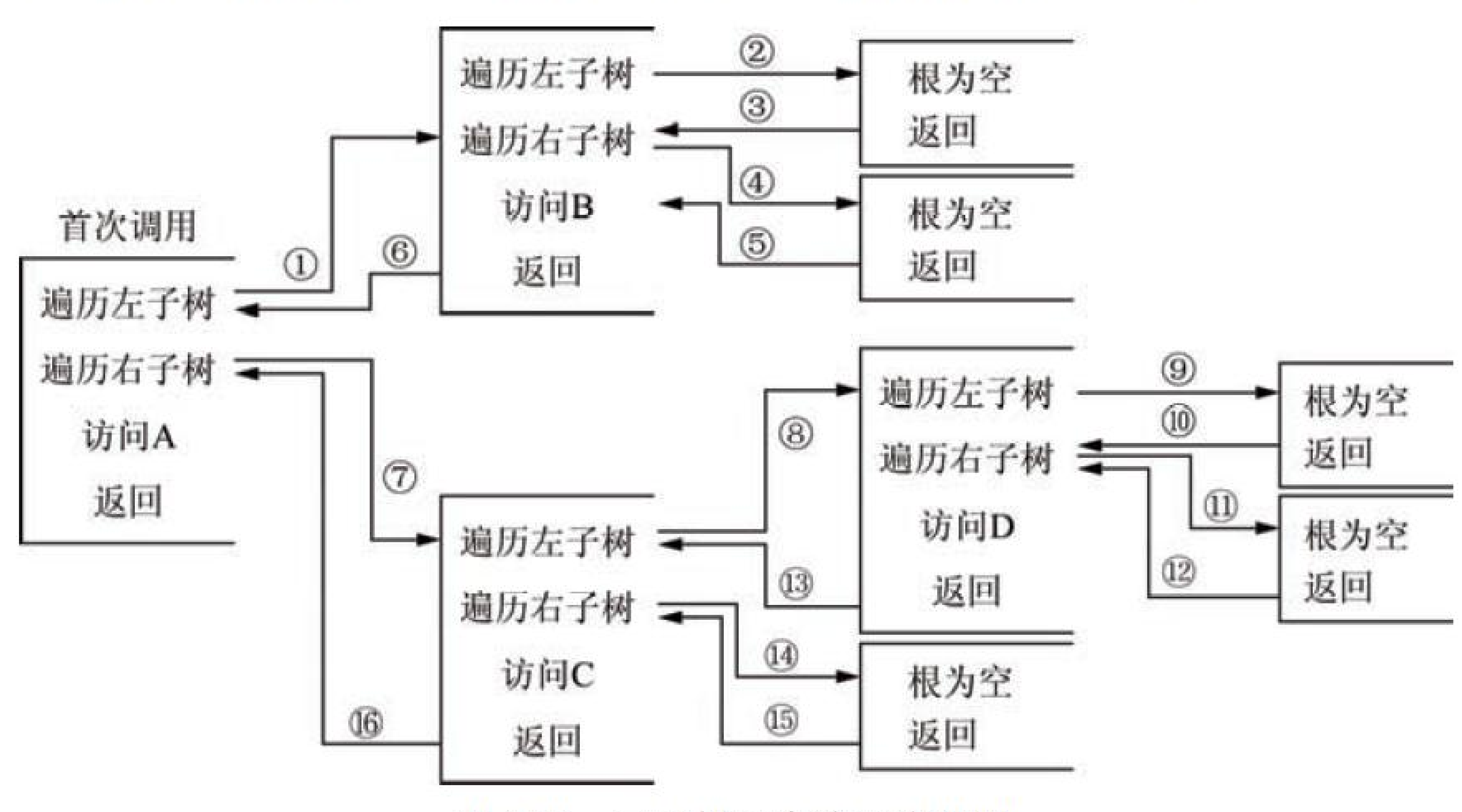


图 3.23　二叉树后序遍历流程图

1）递归算法

二叉树的二叉链表存储结构后序遍历递归算法伪代码描述如下。

```
void postOrderBiTree (BiTree T , visit ()) {
    if(T) {
        postOrderBiTree (T->lChild, visit);
        visit(T->data);
        postOrderBiTree (T->rChild,visit);
    }
}
```

2）非递归遍历算法

后序遍历中，二叉树的根结点在其左右子树均遍历完成后才能被访问，所以其非递归算法较前两种遍历的非递归算法复杂。除前两个非递归算法所需的栈以外，还需要另外的辅助栈记录经过某根结点的次数。

二叉树的二叉链表存储结构后序遍历非递归算法伪代码描述如下。

```
// 利用数组实现栈，仅在 top 下标处增减元素。top=-1 为空栈
void fpostOrderBiTree (BiTree T) {
      BiNode  *s[maxTreeSize],*q;
```

```
        int tempTreeNode[maxTreeSize];
        unsigned nonEmptyStack=1, top=-1;
        q=T;
        do {
            while (q!=NULL) {
                s[++top]=q;
                tempTreeNode[top]=1;          // 第 1 次经过
                q=q->lChild;
            }
            if (top==0)                       // 栈空，后序遍历结束，退出循环
                nonEmptyStack=0;
            else{                             // 第 2 次及第 3 次经过栈的操作
                if(tempTreeNode[top]==1) {
                    tempTreeNode[top]=2;      // 第 2 次经过，不出栈
                    q=s[top];
                    q=q->rChild;
                }else {
                    q=s[top];                 // 第 3 次经过，出栈并且访问结点
                    tempTreeNode[top]=0;
                    top--;
                    printf(q->data);
                    q=NULL;
                }
            }
        }while nonEmptyStack;
    }
```

4．二叉树遍历练习

练习 1 图 3.24 为常见算术表达式对应的二叉树，给出其三种遍历序列，并分析遍历过程。

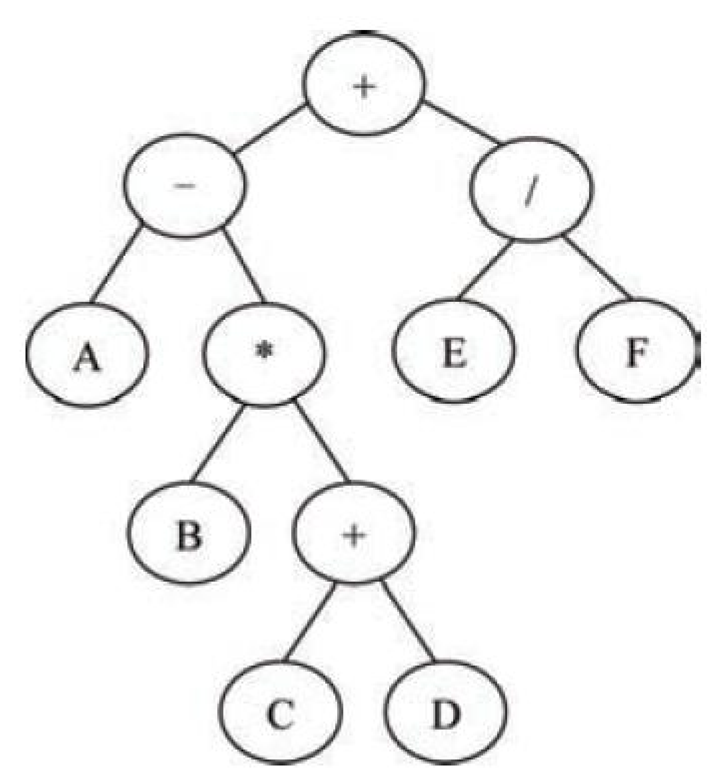


图 3.24 某算术表达式对应的二叉树

【分析】

1．先序序列

1）输出根结点的值 “+”

2）先序遍历根结点“+”的左子树

（1）输出“+”的左子树根结点的值“-”。

（2）先序遍历“-”的左子树。

① 输出“-”的左子树的根结点“A”。

② 先序遍历“A”的左子树，“A”的左子树为空，返回。

③ 先序遍历“A”的右子树，“A”的右子树为空，返回。

（3）先序遍历“-”的右子树。

① 输出“-”的右子树的根结点“*”。

② 先序遍历“*”的左子树。

- 输出“*”的左子树的根结点“B”。
- 先序遍历“B”的左子树，“B”的左子树为空，返回。
- 先序遍历“B”的右子树，“B”的右子树为空，返回。

③ 先序遍历“*”的右子树。

- 输出“*”的右子树的根结点“+”。
- 先序遍历“+”的左子树，输出“+”的左子树的根结点“C”。首先先序遍历“C”的左子树，“C”的左子树为空，返回。然后先序遍历“C”的右子树，“C”的右子树为空，返回。
- 先序遍历“+”的右子树，输出“+”的右子树的根结点“D”。首先先序遍历“D”的左子树，“D”的左子树为空，返回。然后先序遍历“D”的右子树，“D”的右子树为空，返回。

3）先序遍历根结点“+”的右子树

（1）输出“+”的右子树根结点的值“/”。

（2）先序遍历“/”的左子树。

① 输出“/”的左子树的根结点“E”。

② 先序遍历“E”的左子树，“E”的左子树为空，返回。

③ 先序遍历“E”的右子树，“E”的右子树为空，返回。

（3）先序遍历“/”的右子树。

① 输出“/”的右子树的根结点“F”。

② 先序遍历“F”的左子树，“F”的左子树为空，返回。

③ 先序遍历“F”的右子树，“F”的右子树为空，返回。

先序序列为+-A*B+CD/EF

2．中序序列

1）中序遍历根结点“+”的左子树

（1）中序遍历以“+”的左子树的左子树根结点“A”为根的二叉树。

① 中序遍历“A”的左子树，“A”的左子树为空，返回。

② 输出“A”。

③ 中序遍历“A”的右子树，“A”的右子树为空，返回。

（2）输出“-”。

（3）中序遍历“–”的右子树。

① 中序遍历以“–”的右子树根结点“*”的左子树根结点“B”为根的二叉树。

- 中序遍历“B”的左子树，“B”的左子树为空，返回。
- 输出“B”。
- 中序遍历“B”的右子树，“B”的右子树为空，返回。

② 输出“*”。

③ 中序遍历以“–”的右子树根结点“*”的右子树根结点”+”为根的二叉树。

- 中序遍历以“+”的左子树根结点“C”为根的二叉树。首先中序遍历“C”的左子树，“C”的左子树为空，返回。输出“C”。然后中序遍历“C”的右子树，“C”的右子树为空，返回。
- 输出“+”。
- 中序遍历以“+”的右子树根结点“D”为根的二叉树。首先中序遍历“D”的左子树，“D”的左子树为空，返回。输出“D”。然后中序遍历“D”的右子树，“D”的右子树为空，返回。

2）输出“+”

3）中序遍历根结点“+”的右子树

（1）中序遍历以“+”的右子树根结点“/”的左子树根结点“E”为根的二叉树。

① 中序遍历“E”的左子树，“E”的左子树为空，返回。

② 输出“E”。

③ 中序遍历“E”的右子树，“E”的右子树为空，返回 。

（2）输出“/”。

（3）中序遍历以“+”的右子树根结点“/”的右子树根结点“F”为根的二叉树。

① 中序遍历“F”的左子树，“F”的左子树为空，返回。

② 输出“F”。

③ 中序遍历“F”的右子树，“F”的右子树为空，返回。

中序序列为 A – B* C+ D +E /F

3. 后序序列

1）后序遍历根结点“+”的左子树

（1）后序遍历以“+”的左子树的左子树根结点“A”为根的二叉树。

① 后序遍历“A”的左子树，“A”的左子树为空，返回。

② 后序遍历“A”的右子树，“A”的右子树为空，返回。

③ 输出“A”。

（2）后序遍历“–”的右子树。

① 后序遍历以“–”的右子树根结点“*” 的左子树根结点“B”为根的二叉树。

- 后序遍历“B”的左子树，“B”的左子树为空，返回。
- 后序遍历“B”的右子树，“B”的右子树为空，返回。

- 输出“B”。

② 后序遍历以“-”的右子树根结点“*”的右子树根结点“+”为根的二叉树。

- 后序遍历以“+”的左子树根结点“C”为根的二叉树。首先后序遍历“C”的左子树，“C”的左子树为空，返回。然后后序遍历“C”的右子树，“C”的右子树为空，返回，输出“C”。
- 后序遍历以“+”的右子树根结点“D”为根的二叉树。首先后序遍历“D”的左子树，“D”的左子树为空，返回。然后后序遍历“D”的右子树，“D”的右子树为空，返回，输出“D”。
- 输出“+”。

③ 输出“-”的右子树根结点“*”。

（3）输出“-”

2）后序遍历根结点“+”的右子树

（1）后序遍历以“+”的右子树根结点“/”的左子树根结点“E”为根的二叉树。

① 后序遍历“E”的左子树，“E”的左子树为空，返回。

② 后序遍历“E”的右子树，“E”的右子树为空，返回。

③ 输出“E”。

（2）后序遍历以“+”的右子树根结点“/”的右子树根结点“F”为根的二叉树。

① 后序遍历“F”的左子树，“F”的左子树为空，返回。

② 后序遍历“F”的右子树，“F”的右子树为空，返回。

③ 输出“F”。

（3）输出“/”。

3）输出“+”

后序序列为：ABCD+*-EF/+。

练习 2　编写创建二叉树的算法。

解析：

（1）利用满二叉树的特点构建二叉树：首先将待构建的二叉树补全为满二叉树，然后对每个结点自上而下、从左至右进行编号，按照编号顺序创建二叉树，具体如下。

```
void createBiTree(BiTree T) {
    BiTNode *t=NULL, *s[maxTreeSize], *p;
    scanf(&i,&x);
    while (i!=0)&&(x!=0) {
        p=(struct BiTNode *)malloc(sizeof(struct BiTNode));
                                                    // 申请一个结点空间
        p->data=x,  p->lChild=NULL,  p->rChild=NULL;
        s[i]=p;
        if (i==1)                                   // p 为根结点
            t=p;
        else {
            j=i/2;                                  // j 为双亲结点编号
            if ((i%2)==0)
                s[j]->lChild=p;
```

```
            else
                s[j]->rChild=p;
        }
        scanf(&i,&x);
    }
}
```

（2）利用二叉树的递归性质创建二叉树：由于二叉树的定义为递归形式，故二叉树的操作用递归方法实现比较简单。

```
void createBiTreeRec(BiTree T) {
    BiTNode *p;
    scanf(&x);
    if(x==0)
        T=NULL;
    else{
        p=(struct BiTNode *)malloc(sizeof(struct BiTNode));
                                                    // 申请一个结点空间
        p->data=x;
        createBiTreeRec(p->lChild);                 // 递归创建左子树
        createBiTreeRec(p->rChild);                 // 递归创建右子树
    }
}
```

练习 3 统计二叉树的叶子结点、度为 1、度为 2 的结点的数量。

解析：设一个含三个元素的数组 n[3]，规定 n[0]代表叶子结点的个数，n[1]代表度为 1 结点的个数，n[2]代表度为 2 的结点个数，初值均为 0，遍历（以先序为例，递归算法）的同时进行各类结点个数的统计。伪代码如下。

```
void countBiTreeNodes(BiTree T, unsigned n[3]) {
    BiTNode *p=T;
    While(p!=NULL){
        if(p->lChild!=NULL && p->rChild!=NULL)
            n[2]++;
        else if(p->lChild==NULL && p->rChild==NULL)
                n[0]++;
            else
                n[1]++;
        countBiTreeNodes (p->lChild);            // 递归统计左子树
        countBiTreeNodes (p->rChild);            // 递归统计右子树
    }
}
```

3.3.5 线索二叉树

由二叉树性质可知，利用二叉链表存储具有 n 个结点的二叉树，需要 n−1 个指针。但 n 个结点共含 2*n 个指针域，故存在 n+1 个空指针。

分别用先序、中序、后序遍历二叉树，均可得到一个线性序列。如果以后再次需要该线性序列，有两种方案：

（1）有需要时，再次遍历，缺点是浪费时间。

（2）保存已经得到的线性序列，后期直接使用。缺点是浪费时间，浪费空间。

可利用二叉链表 d 的 n+1 个空指针简化算法。即如果某结点存在左孩子，则该结点的左指针指向其左孩子，否则指向其前驱（按照某种遍历策略得到的线性序列）；如果某结点存在右孩子，则该结点的右指针指向其右孩子，否则指向其后继（按照某种遍历策略得到的线性序列）。这个过程称为对二叉树按照某种策略进行线索化。图 3.25 为一棵后序线索二叉树。

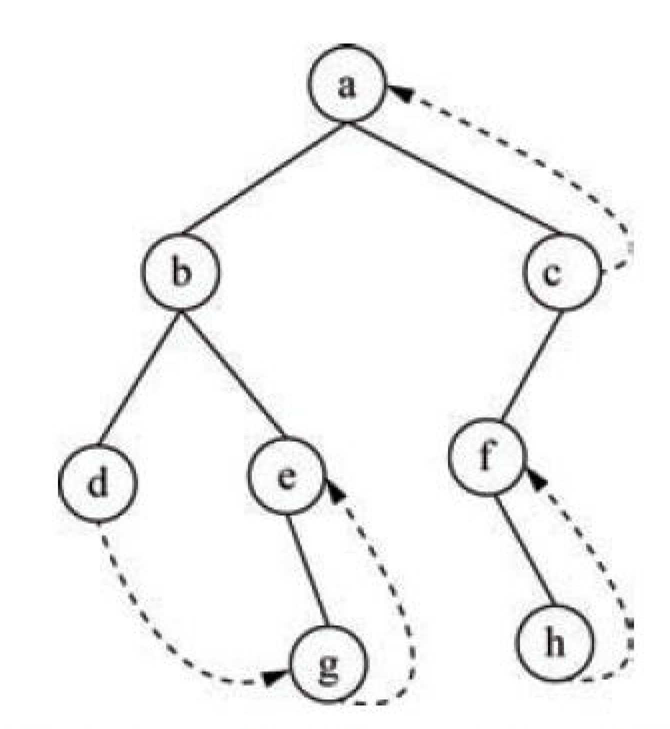


图 3.25　后序线索二叉树示意图

二叉树线索化流程如下。

（1）按照某种策略遍历二叉树。

（2）对各结点依次进行如下处理。

① 无左孩子，将其左链域的值修改为其前驱结点的地址。

② 无右孩子，将其右链域的值修改为其后继结点的地址。

③ 对遍历序列的首元素的左链域和最后一个元素结点的右链域进行处理。

对二叉树进行线索化后，再次需要该二叉树的遍历序列时，只需找到该遍历的第一个元素，然后依次找各个结点的后继即可。

线索二叉树找结点的后继的流程如下。

（1）先序、中序线索二叉树比较容易。

（2）后序线索树中找后继结点包含以下三种情况。

① 结点 x 为二叉树的根，则其后继为空。

② 结点 x 为其双亲的右孩子，或为其双亲的左孩子且其双亲没有右子树，则其后继为其双亲结点。

③ 结点 x 为其双亲的左孩子，且其左孩子有右子树，则其后继为双亲的右子树上按后继的右子树上按后序遍历列出的第一个结点。

能够快速找到结点后继的存储结构为三叉链表，每个结点包含四个域：

- 值域。
- 指向其左子树根结点的指针。
- 指向其右子树根结点的指针。
- 指向其双亲结点的指针。

3.3.6 二叉排序树

二叉排序树为特殊二叉树，主要用于查找，所以又称为二叉查找树。由于其中序遍历为有序序列，所以二叉排序树通常仅进行中序遍历。本节仅介绍二叉排序树的基本概念和创建、插入、删除等基本操作，查找操作在第 5 章介绍。

1. 定义

二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

（1）若左子树不空，则左子树上所有结点的值均小于其父结点的值。

（2）若右子树不空，则右子树上所有结点的值均大于其父结点的值。

（3）左右子树分别为二叉排序树。

由二叉排序树的上述定义可知，中序遍历二叉排序树可得递增有序序列。

若规定左子树上所有结点的值均大于其父结点的值，右子树上所有结点的值均小于其父结点的值，也构成二叉排序树，中序遍历可得递减有序序列。

二叉排序树中插入/删除一个结点后，必须保证操作之后仍保持二叉排序树特征。

2. 二叉排序树的构造与插入结点

1）过程

（1）若二叉排序树为空，则待插入结点为根结点。

（2）若二叉排序树非空，则待插入结点为叶子结点：

① 待插入结点的值和根结点的值进行比较。

② 如果小于，则在左子树进行如步骤①的比较；

③ 如果大于，则在右子树进行如步骤①的比较。

④ 重复步骤①～③直至某左子树或右子树为空，则待插入结点为左子树或右子树根结点（二叉排序树的叶子结点）。

2）伪代码（设二叉排序树不含重复值结点）

```
int  insertBalanceSortedBiT(BiTree  T,  eleType  e){
   if(!T){                                      // 二叉排序树为空
         s=(BiTree)malloc(sizeof(BiNode));
         s->data=e;
         s->lchild=s->rchild=NULL;
         T=s;
         return 1;
      }
      if(e == T->data )
         return  0;                             // 已包含该关键字，插入失败
      if(( e < T->data )
         insertBalanceSortedBiT(T->lChild, e);
      else
         insertBalanceSortedBiT(T->rChild, e);
}
```

3）示例

为序列{65,23,50,16,42,22,90,20,12,60}构造二叉排序树，如图 3.26（a）～图 3.26

（k）所示。

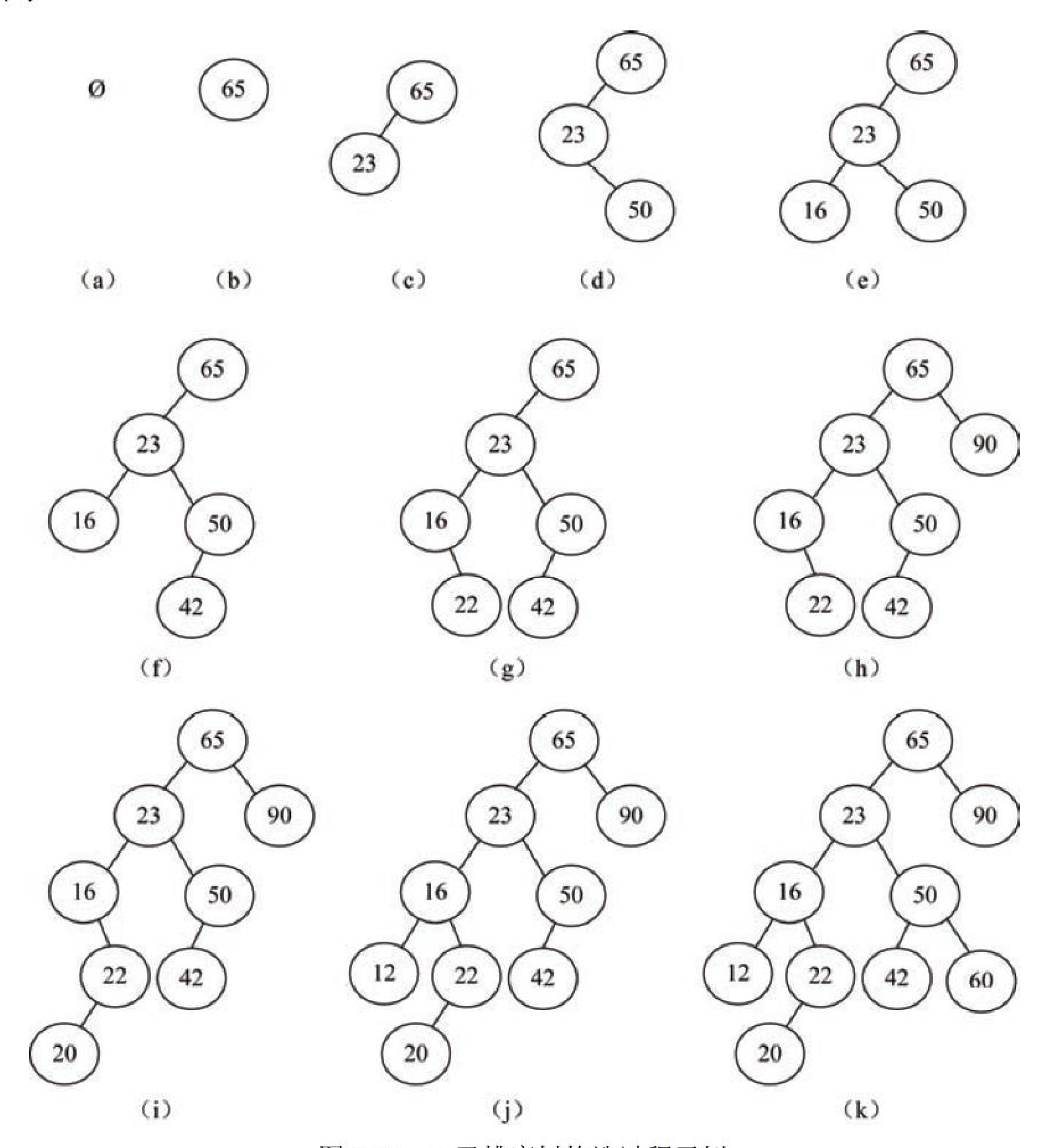


图 3.26　二叉排序树构造过程示例

3．二叉排序树删除结点

1）过程

假设待删结点为 p，其父结点为 f，需考虑以下 3 种情况：

（1）p 为叶子结点

① p 为 f 的左子树：f->lChild =NULL。

② p 为 f 的右子树：f->rChild=NULL。

（2）p 只有一个子树

① p 只有左子树：用 p 的左孩子代替 p，释放 p 所占存储空间。二叉排序树的中序遍历序列中，没删结点的相对位置不变，如图 3.27（a） 和图 3.27（b）所示。

② p 只有右子树，用 p 的右孩子代替 p，释放 p 所占处处空间。二叉排序树的中序遍历序列中，没删结点的相对位置不变。如图 3.28（a） 和图 3.28（b）所示。

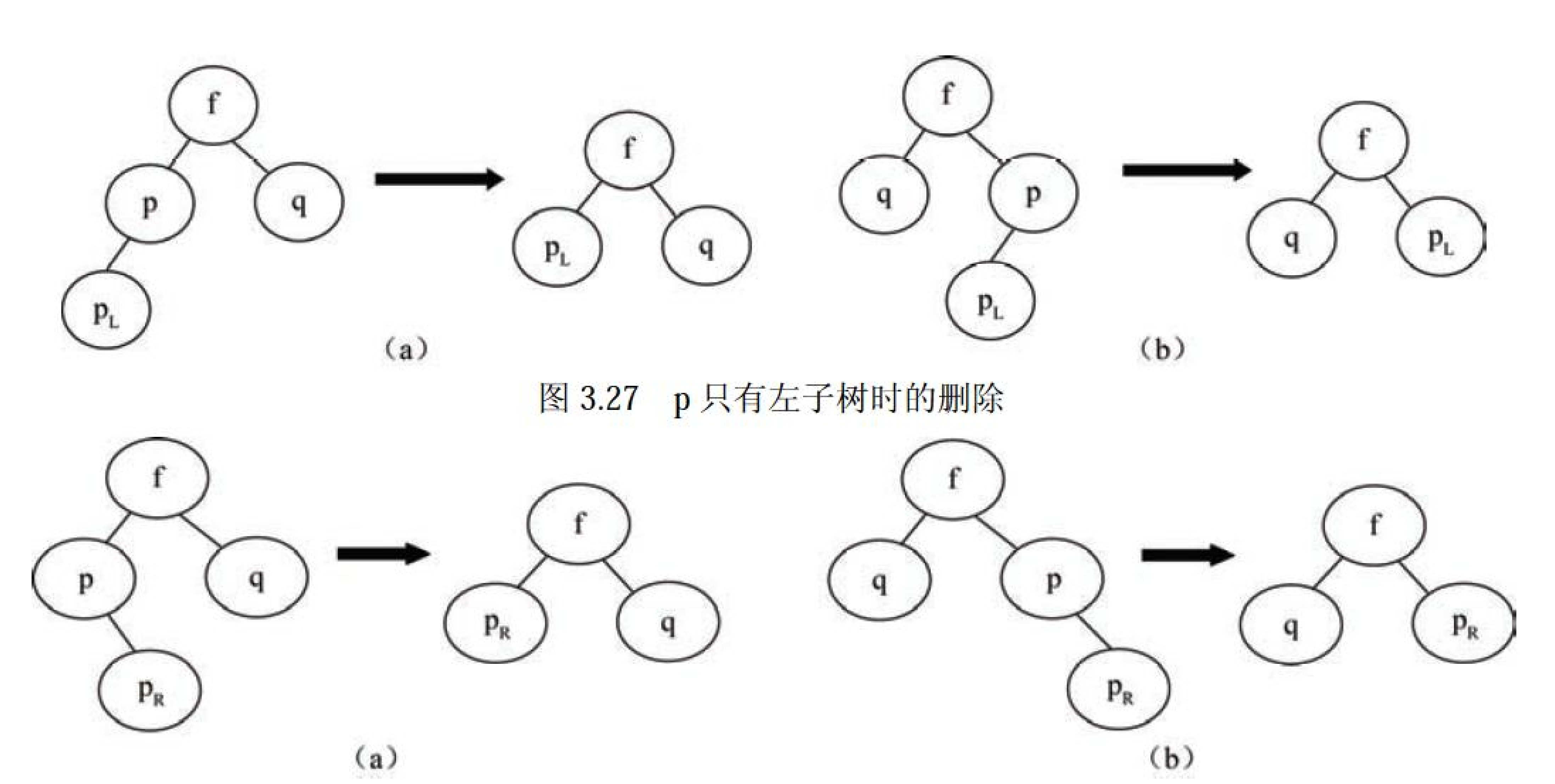


图 3.27　p 只有左子树时的删除

图 3.28　p 只有右子树的删除

（3）p 有两个子树。

① 用 p 的直接前驱代替 p，释放 p 所占存储空间。二叉排序树的中序遍历序列中，没删结点的相对位置不变。如图 3.29（a）～图 3.29（c）所示。

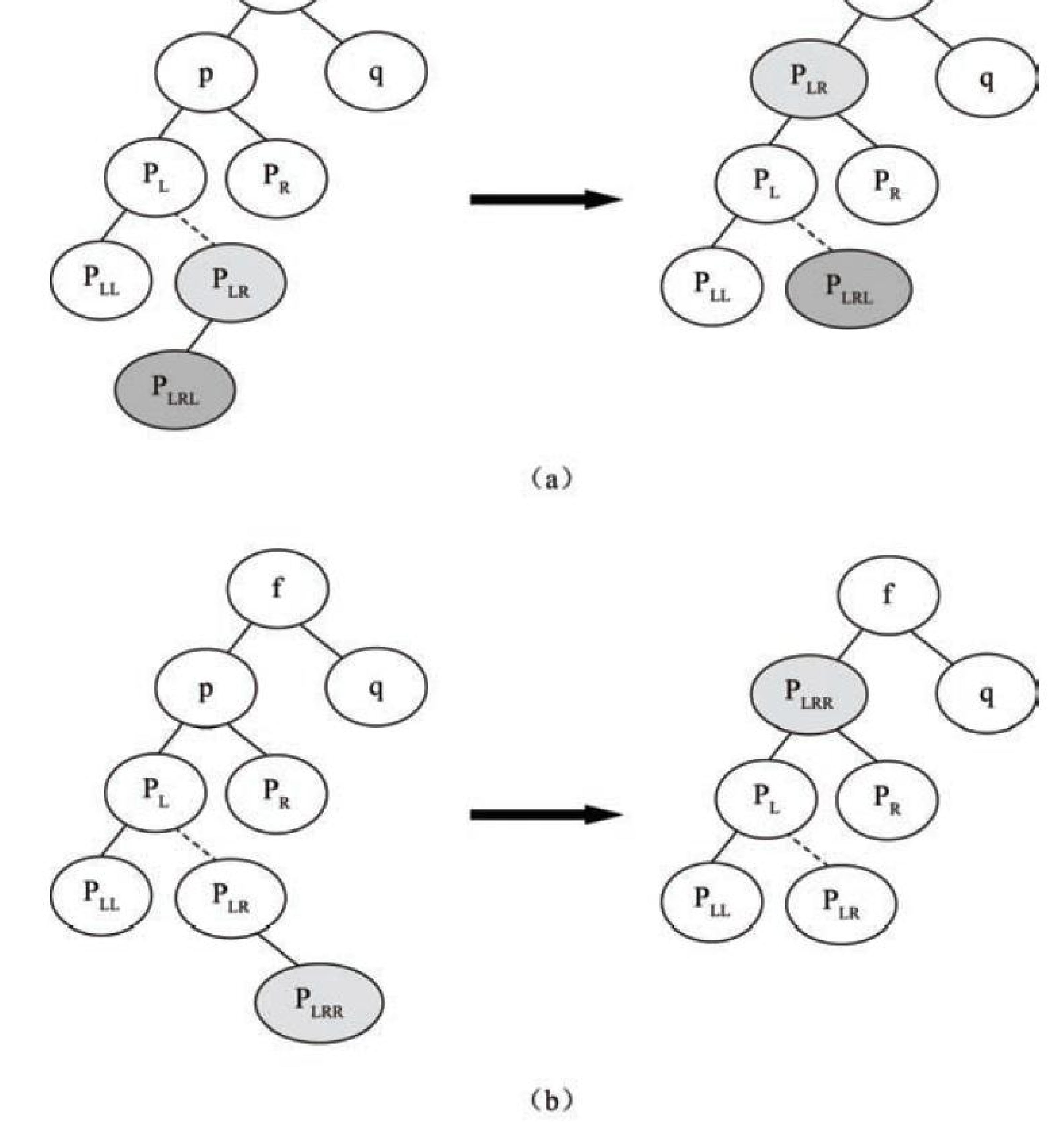
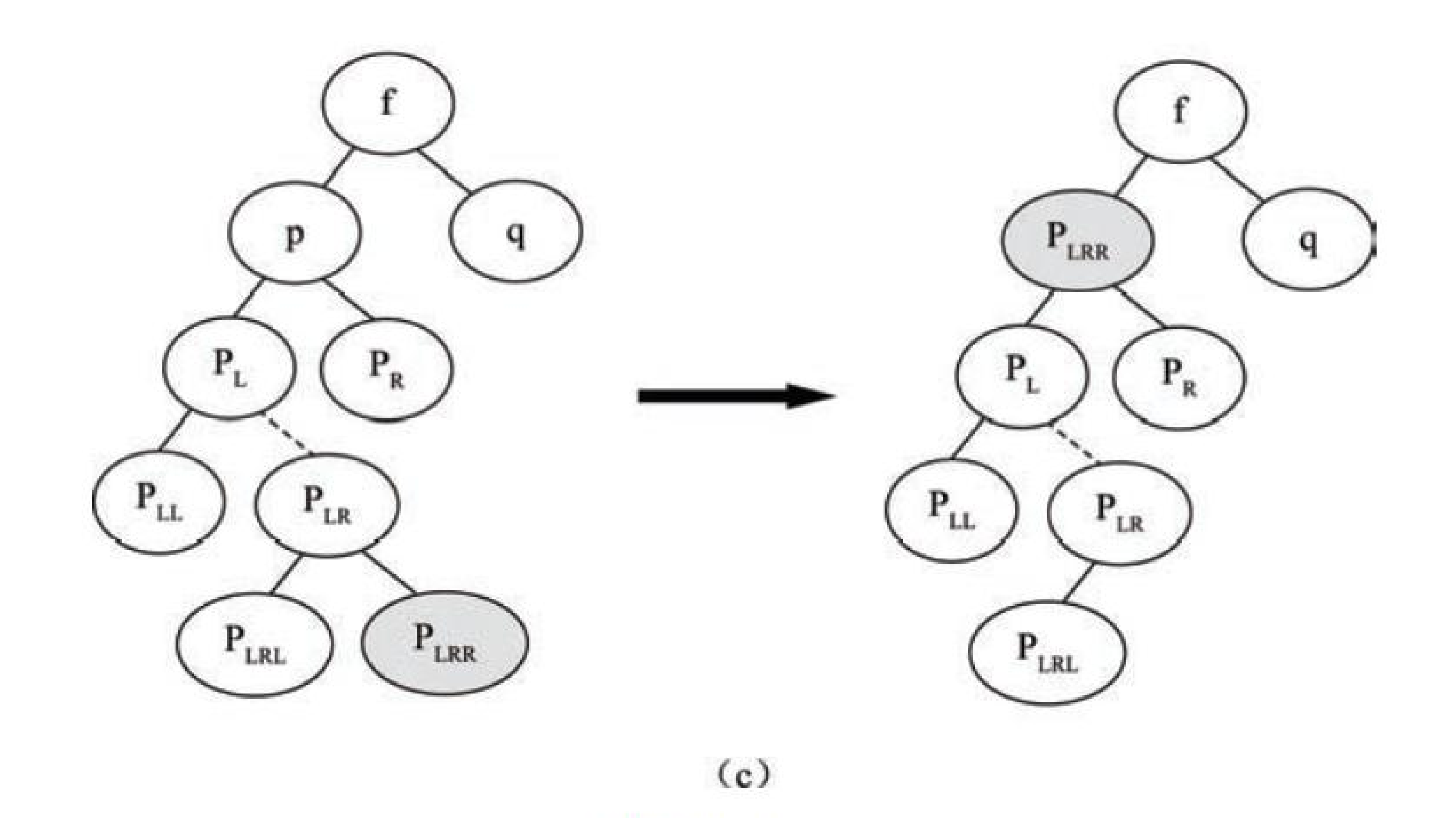


图 3.29　用 p 的直接前驱代替 p



（c）

图 3.29（续）

② 用 p 的直接后继代替 p，释放 p 所占存储空间。二叉排序树的中序遍历序列中，没删结点的相对位置不变，如图 3.30 所示。

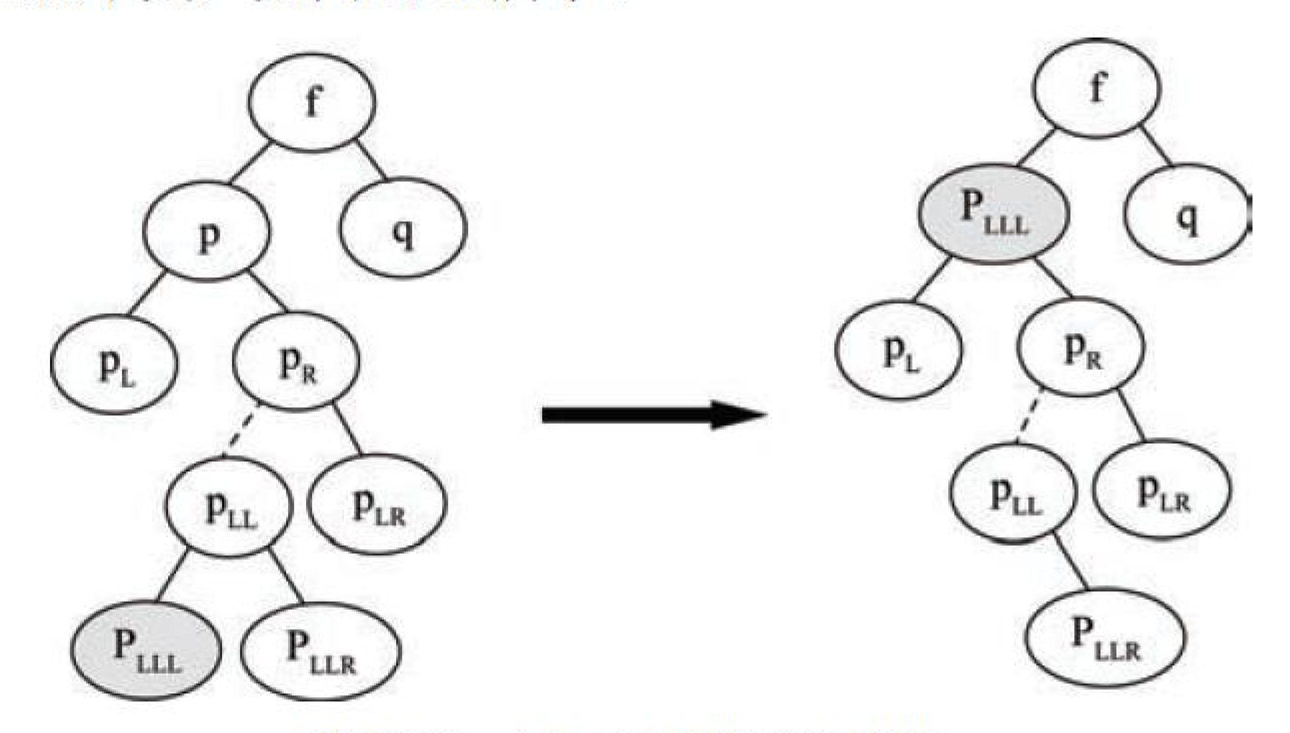
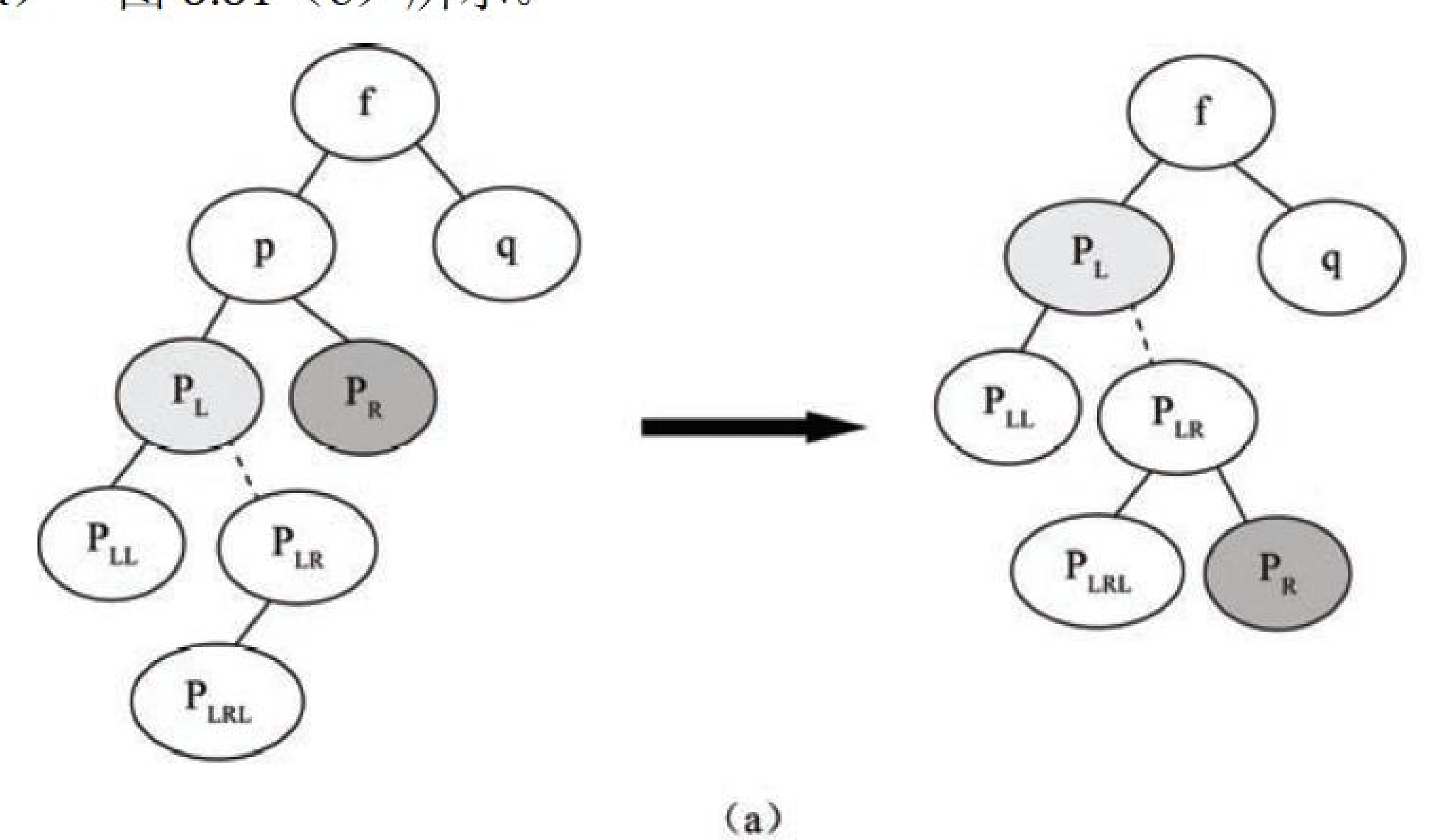


图 3.30 用 p 的直接后继代替 p

③ 让 p 的左子树为 f 的左子树，p 的右子树为 p 的左子树最右端的结点的右子树，释放 p 所占存储空间。二叉排序树的中序遍历序列中，没删结点的相对位置不变，如图 3.31（a）～图 3.31（c）所示。



（a）

图 3.31 p 的左子树为 f 的左子树，p 的右子树为 p 的左子树最右端的结点的右子树

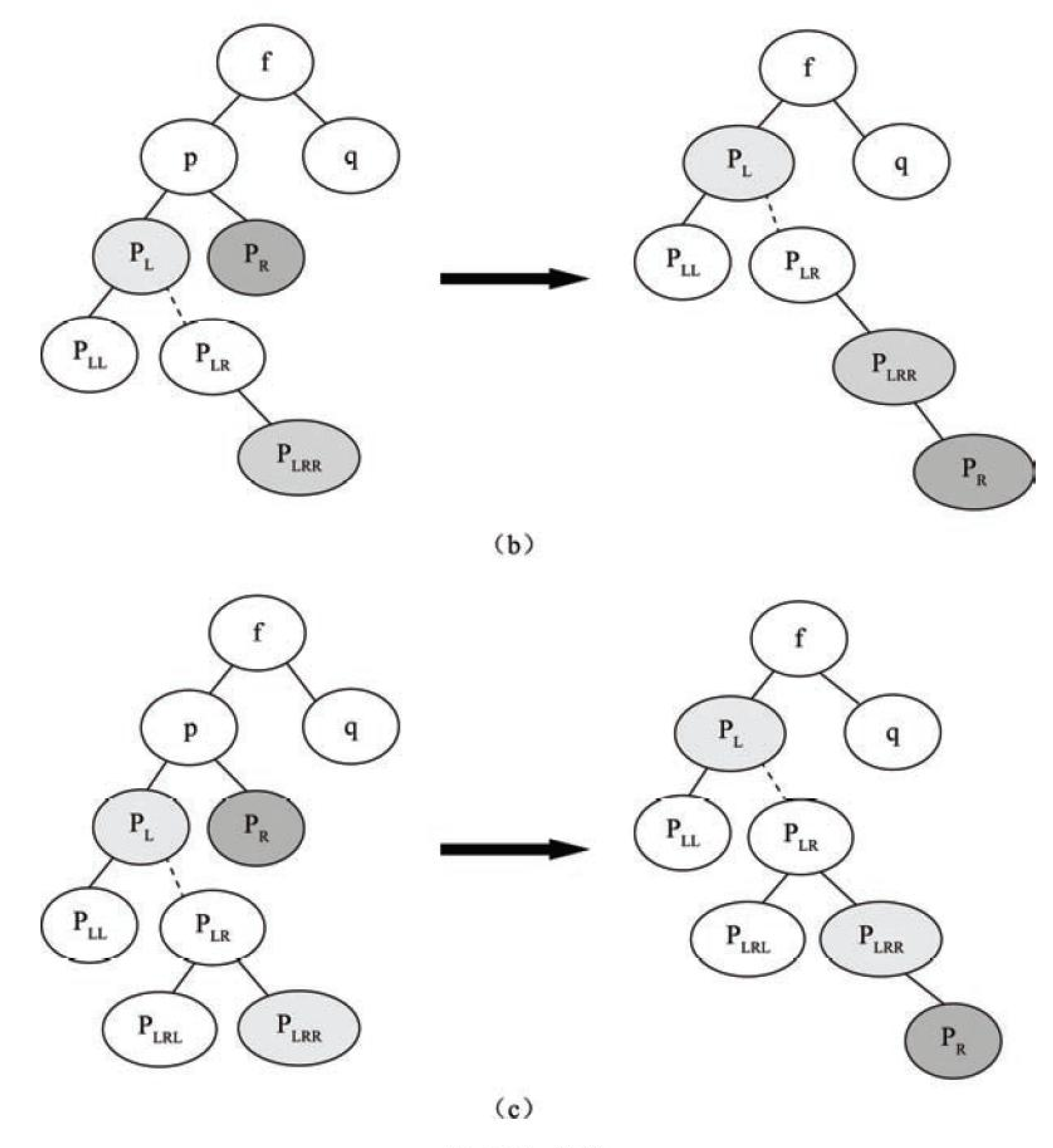


图 3.31（续）

2）示例

删除图 3.26（k）二叉排序树值为 23 的结点，结果如图 3.32 所示。

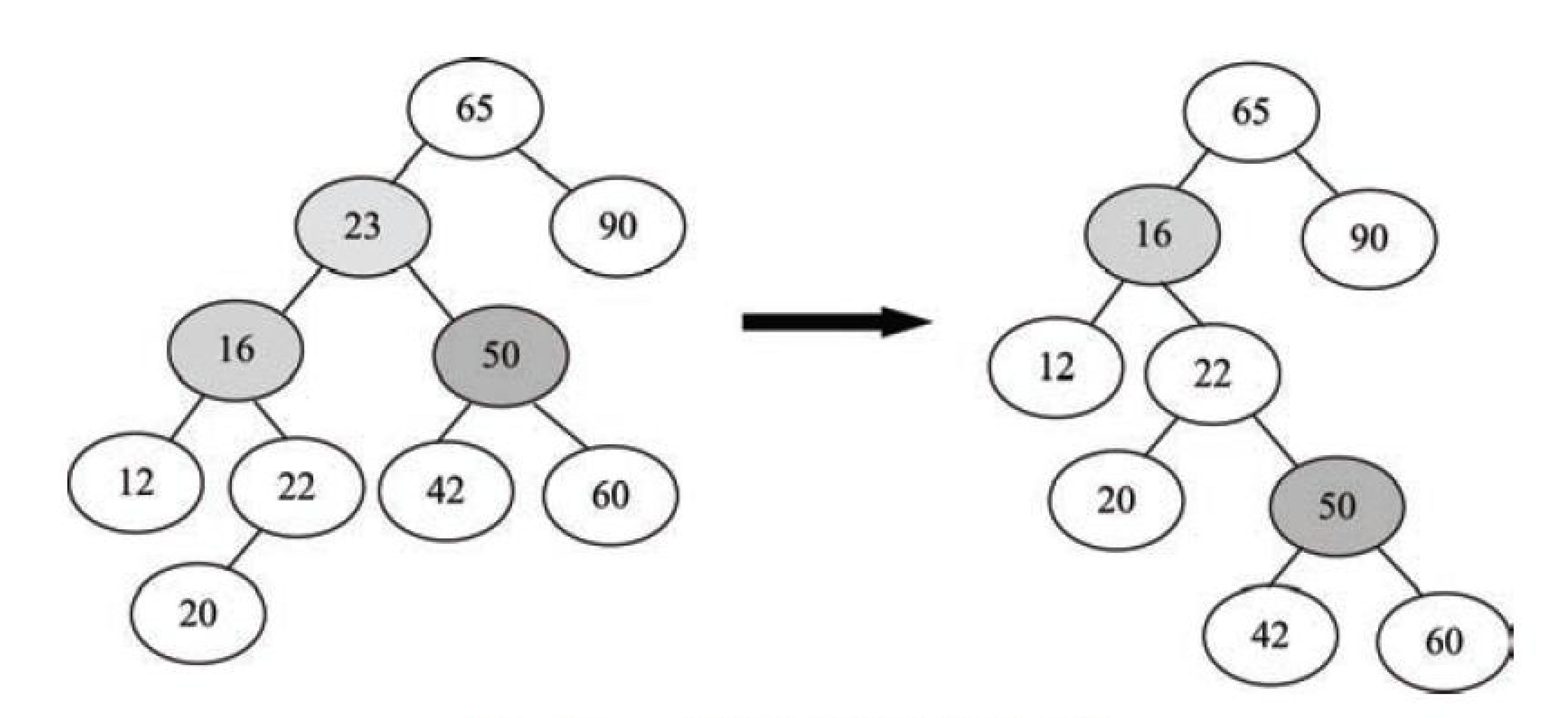


图 3.32　二叉排序树删除结点示例

3.3.7　平衡二叉树

平衡二叉树又称 AVL 树，为同等条件下、高度或深度最小的二叉排序树。

1．定义

平衡二叉树可以是一棵空树，或具有下列性质的二叉树：

- 其左右子树都是平衡二叉树。
- 左、右子树的深度之差的绝对值不超过 1。

2．平衡因子

平衡因子（Balance Factor，BF）是二叉树中某个结点的左子树深度减其右子树深度的差。平衡二叉树中任何结点的平衡因子只可能是-1、0、1。

AVL 树中插入新结点后有可能失衡，此时必须重新调整树结构，使之恢复平衡。恢复平衡只需对最小不平衡子树进行处理。可通过旋转完成平衡，旋转有如下四种情况。

（1）LL 平衡旋转：若在 AVL 树中 A 结点的左子树的左子树上插入结点，使 A 的平衡因子从 1 增加至 2，则需要以 A 左子树根结点 B 为轴进行一次顺时针旋转来进行平衡化，如图 3.33 所示。

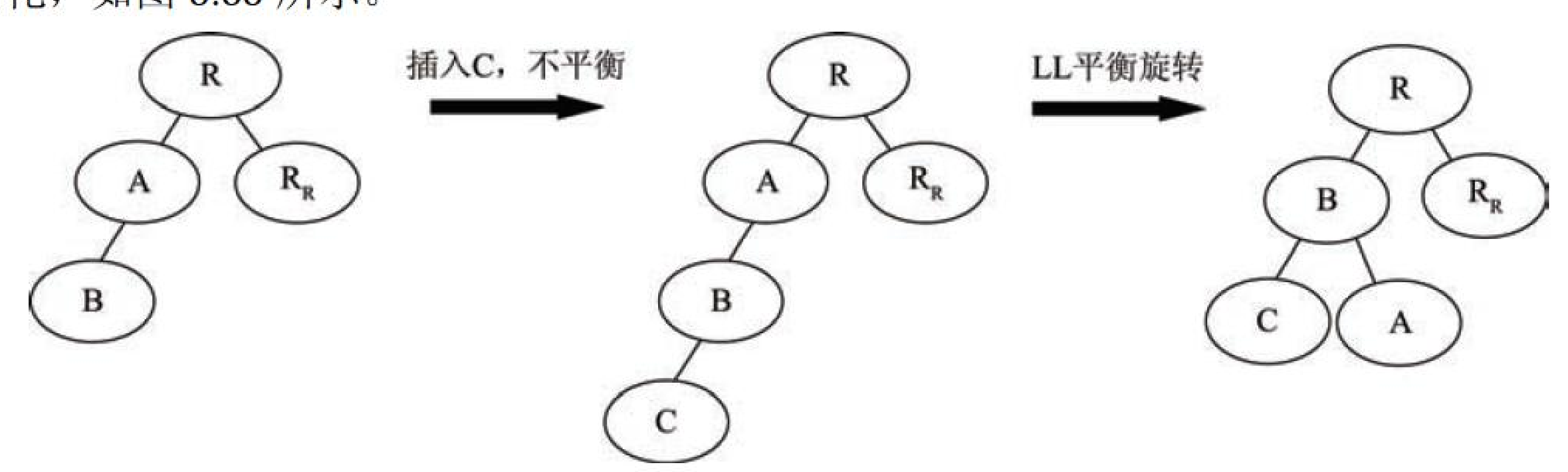


图 3.33　LL 型平衡旋转

注意：原来此子树的高度为 2，调整后仍为 2。所以，LL 平衡旋转不会改变其他结点的 BF。

（2）RR 平衡旋转：若在 AVL 树中 A 结点的右子树的右子树上插入结点，使 A 的平衡因子从 -1 减小至-2，则需要以 A 右子树根结点 B 为轴进行一次逆时针旋转来进行平衡化，如图 3.34 所示。注意，原来此子树的高度为 2，调整后仍为 2。所以，RR 平衡旋转不会改变其他结点的 BF。

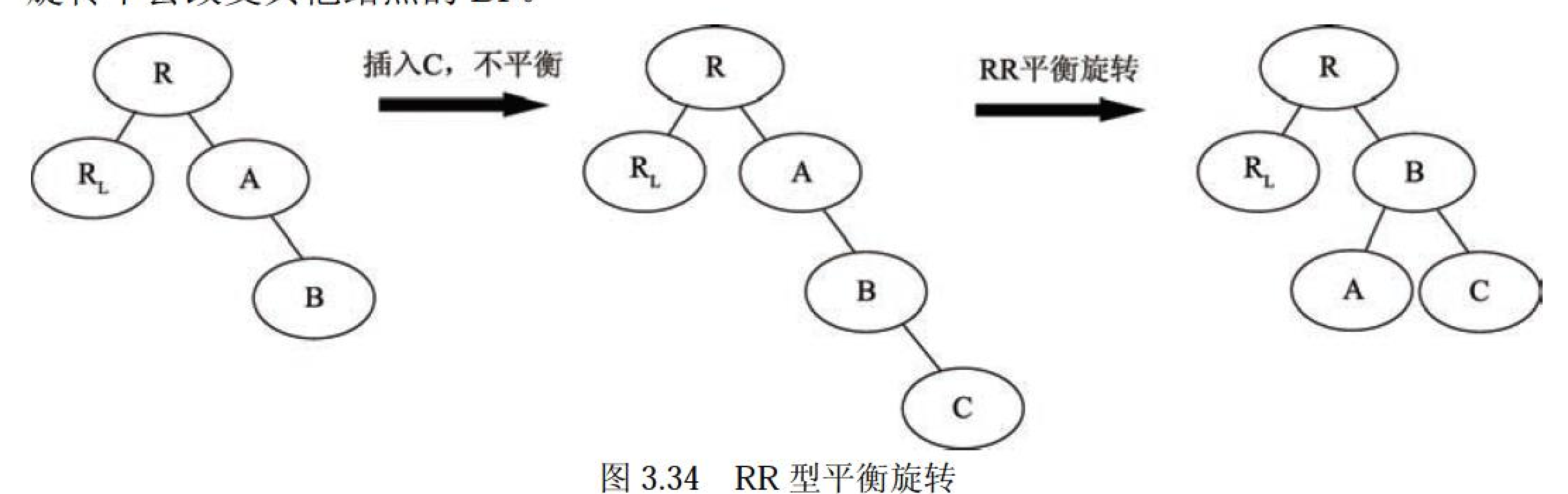


图 3.34　RR 型平衡旋转

（3）LR 平衡旋转：若在 AVL 树中 A 结点的左子树的右子树上插入结点，使 A 的平衡因子从 1 增加至 2，则需要以新插入结点 C 为轴先进行一次逆时针旋转，再进行一次顺时针旋转，进行平衡化，如图 3.35 和图 3.36 所示。注意，原来此子树的高度为 2，调整后仍为 2。所以，RR 平衡旋转不会改变其他结点的 BF。

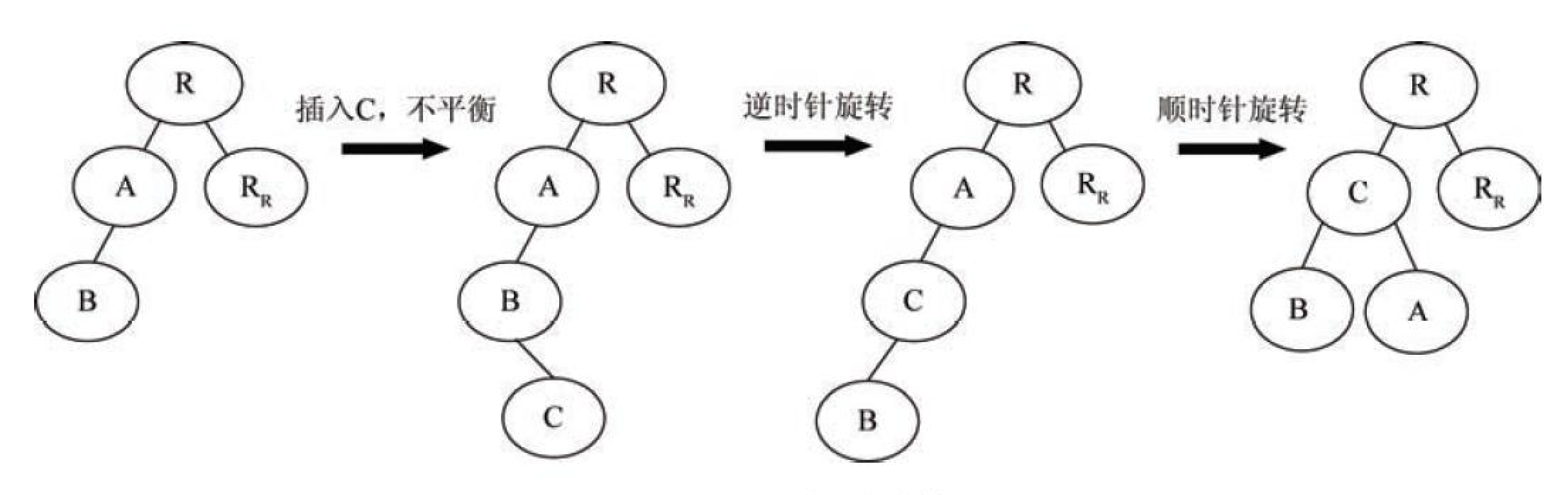


图 3.35　LR 型平衡旋转 1

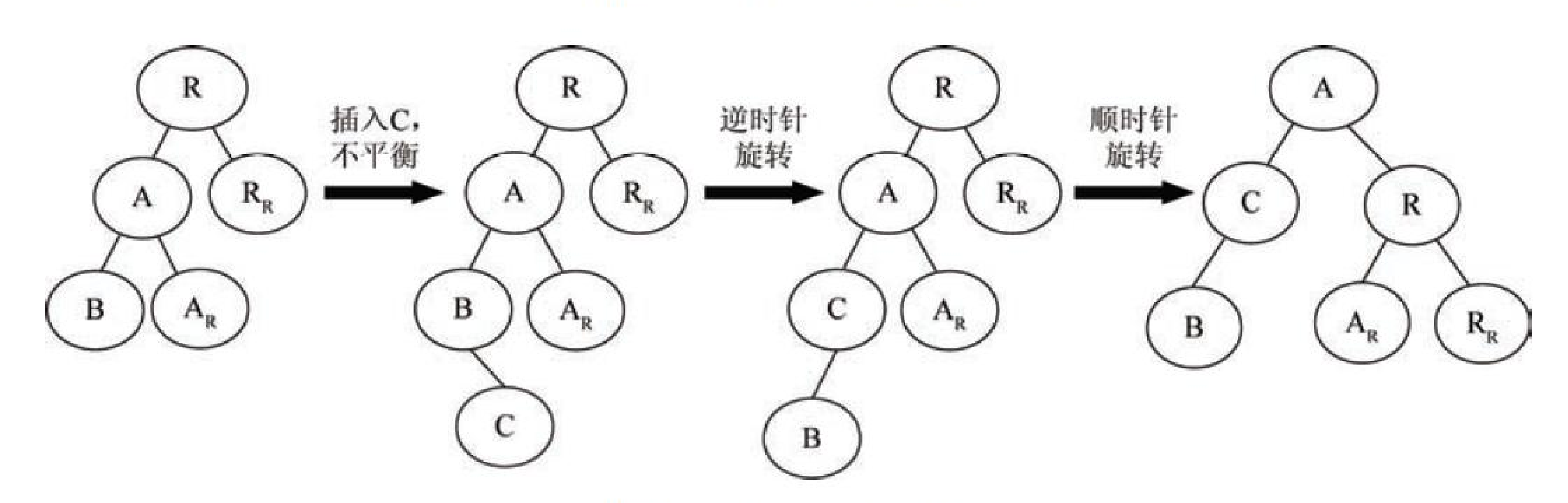


图 3.36　LR 型平衡旋转 2

（4）RL 平衡旋转：若在 AVL 树中 A 结点的右子树的左子树上插入结点，使 A 的平衡因子从 -1 减小至-2，则需要以新插入结点 C 为轴先进行一次顺时针旋转，再进行一次逆时针旋转，进行平衡化，如图 3.37 和图 3.38 所示。注意，原来此子树的高度为 2，调整后仍为 2。所以，RR 平衡旋转不会改变其他结点的 BF。

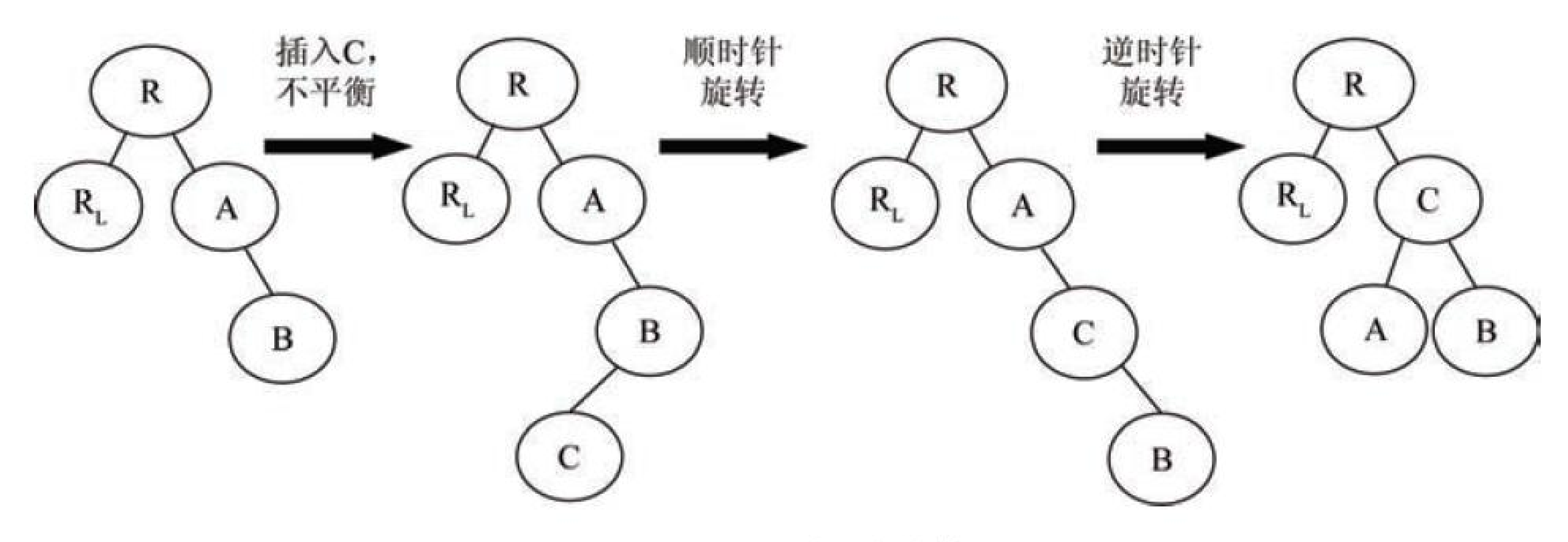


图 3.37　RL 型平衡旋转 1

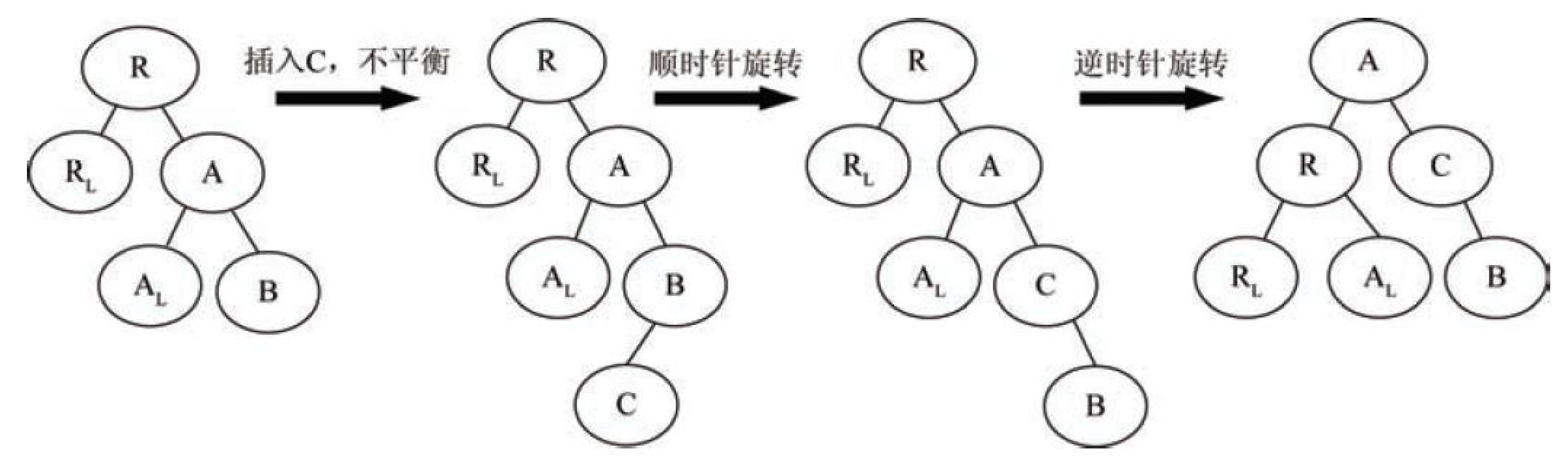


图 3.38　RL 型平衡旋转 2

3．示例

为序列{65,23,50,16,42,22,90,20,12,60}构造平衡二叉排序树，如图 3.39（a）～图 3.39（m）所示。

（a）　（b）　（c）

（d）

（e）　（f）

（g）　（h）　（i）

图 3.39　平衡二叉排序树构造过程示例

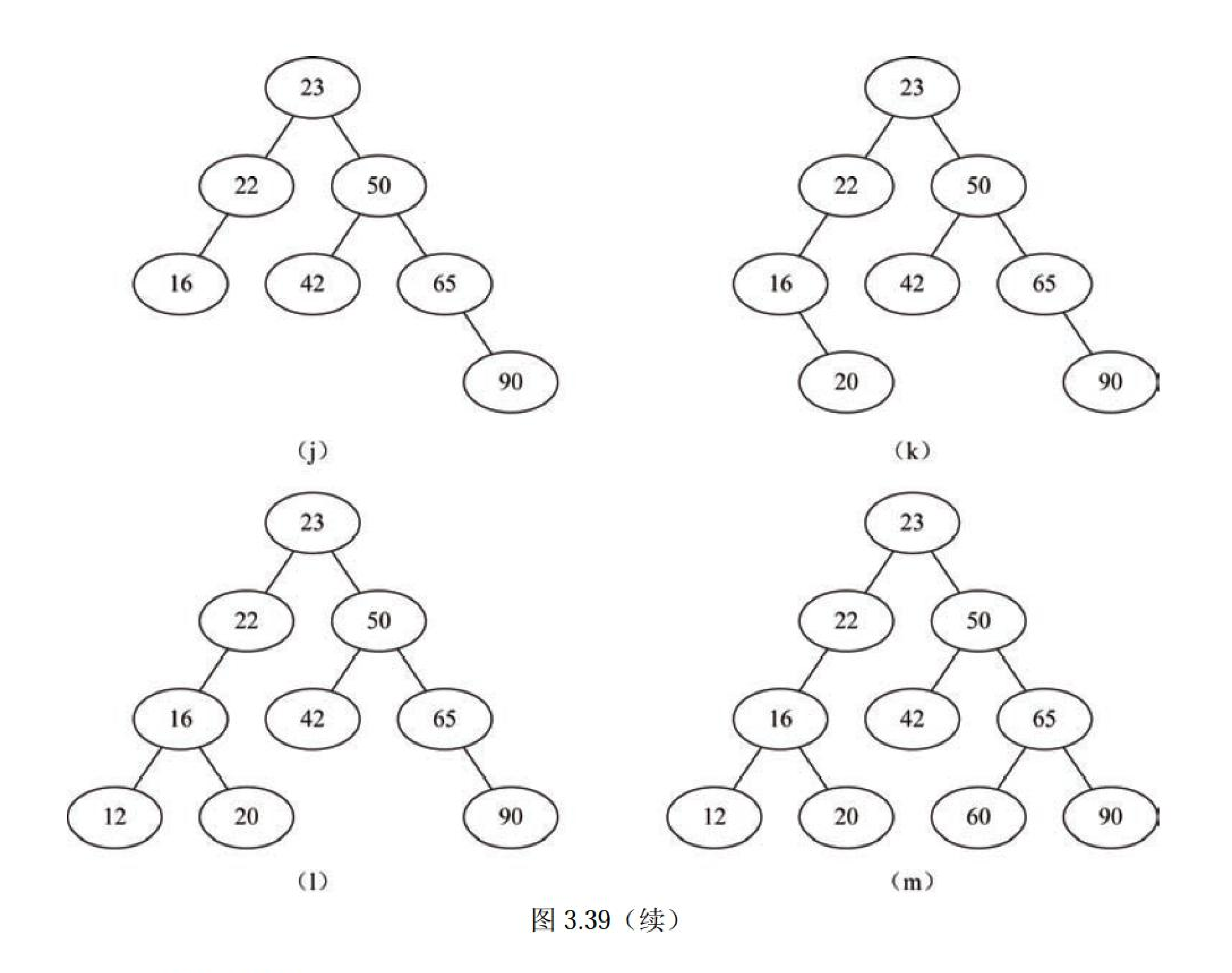


图 3.39（续）

3.3.8 哈夫曼树

1．基本概念

（1）路径长度：从树的一个结点到另一个结点之间的分支构成这两个结点之间的路径，路径上的分支数称为路径长度。

（2）树的路径长度：树的路径长度是从树根到每一个结点的路径长度之和。

（3）结点的带权路径长度：从该结点到树根之间的路径长度与结点的权值的乘积。

（4）树的带权路径长度：所有叶子结点的带权路径长度之和，用 WPL 表示。

（5）哈夫曼（Huffman）树：带权路径长度最小的二叉树，又称最优二叉树。注意，哈夫曼（Huffman）树可能不唯一。

（6）无前缀编码：对一系列字符，例如 a、b、c…进行编码，其中任何一个字符的编码都不是其他字符编码的前缀。比如，若字符 a 的编码为 001，则其他任何字符的编码都不以 001 打头，具有这种特征的编码称为无前缀编码。

2．哈夫曼编码

哈夫曼树的重要应用常见于通信电文字符编码。由于二叉树的根到各个叶子结点的路径均不相同，可利用这些不同的路径表示各叶子结点的编码。

1）哈夫曼编码过程

（1）以待编码字符为叶子结点生成一棵哈夫曼树。

（2）按照左 0 右 1（或左 1 右 0）为哈夫曼树的所有边编码。

（3）从根结点到每个叶子结点边编码组成的编码称为该叶子结点的哈夫曼编码。

2）哈夫曼树构造流程

（1）将 n 个给定的权值进行升序排列，结果为 $w=\{w_1, w_2, \cdots, w_n\}$。

（2）从 w 中选取前两个权值 w_1, w_2，以其为叶子结点构建一棵二叉树，该二叉树根结点的权值为 $w_{12} = w_1+w_2$。将 w_{12} 并入 w，并保持 w 的升序性质不变。

（3）重复（2），直至 w 仅含一个权值。

3．示例

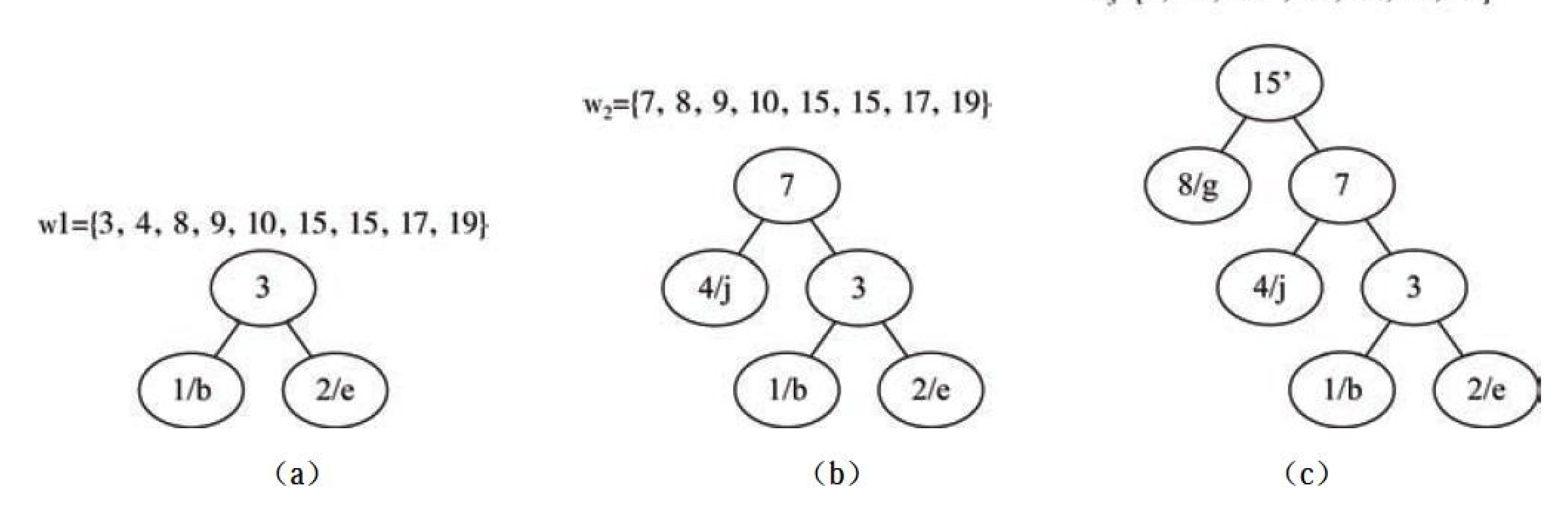
假设某通信电文由 a～j 共 10 个字母组成，出现频率分别为 19%、1%、15%、9%、2%、10%、8%、15%、17%、4%，对其进行哈夫曼编码。

解：

（1）分别把频率×100 作为权值，并按升序排列。

$$w=\{1，2，4，8，9，10，15，15，17，19\}。$$

（2）构造哈夫曼树，如图 3.40（a）～图 3.40（i）所示。



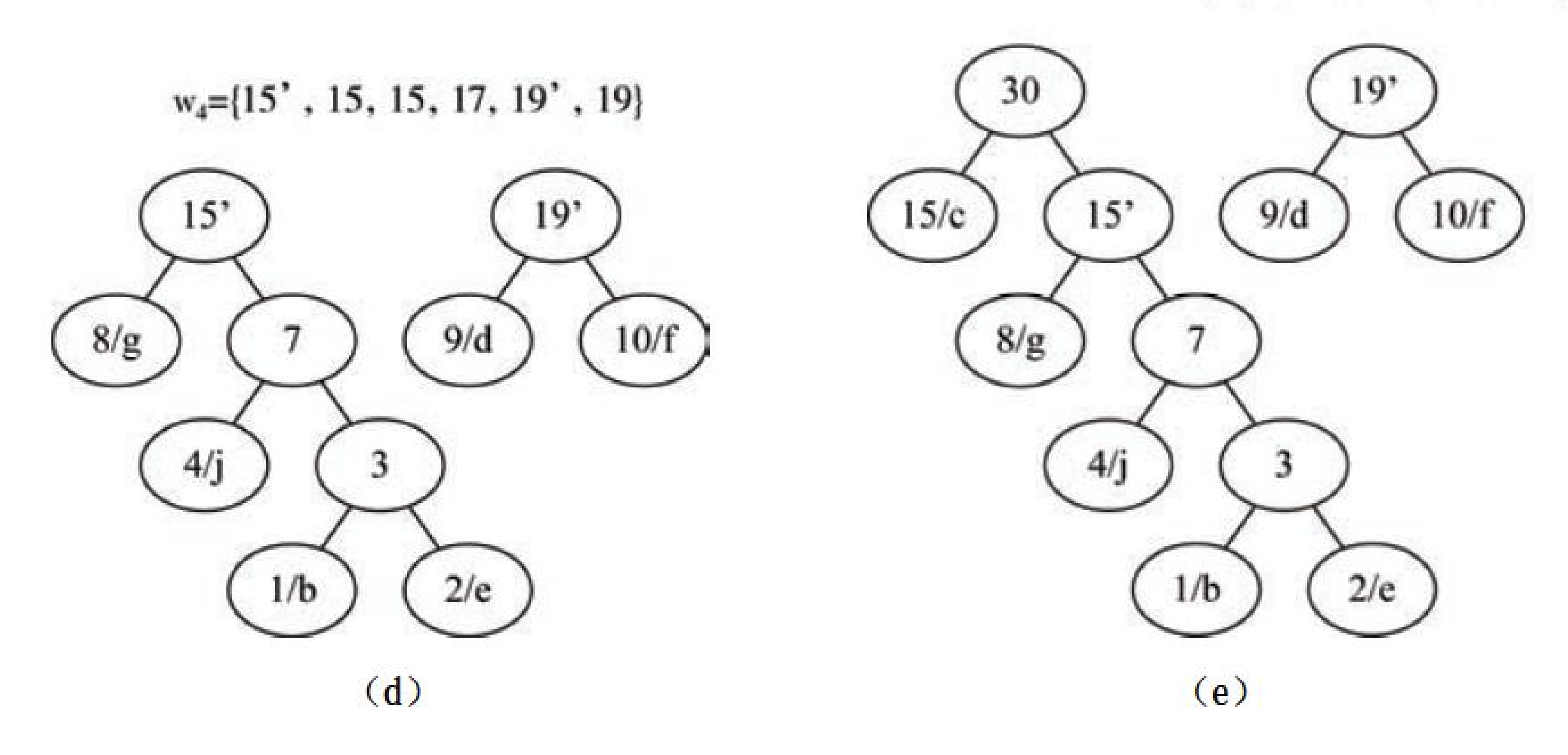


图 3.40　哈夫曼树构造过程示例

w_6={19’, 19, 30, 32}

（f）

w_7={30, 32, 38}

（g）

w_8={62, 38}

（h）

w_9={100}

（i）

图 3.40（续）

（3）叶子结点进行哈夫曼编码，如图 3.41 所示（以左 0 右 1 为例）。

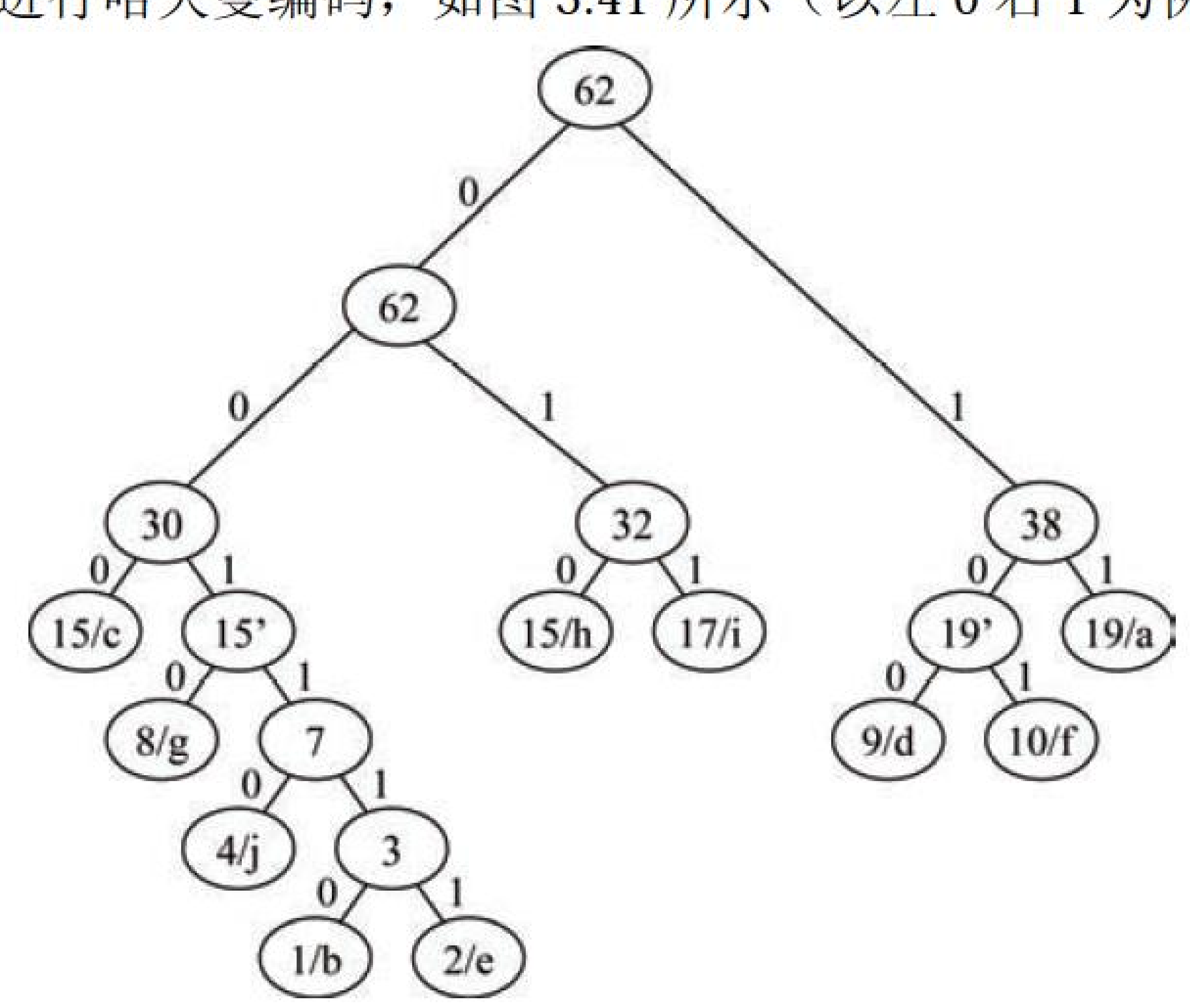


图 3.41　哈夫曼编码示例

编码结果如表 3.1 所示。

表 3.1 哈夫曼编码

字符	哈夫曼编码	频率/%	码长
a	11	19	2
b	001110	1	6
c	000	15	3
d	100	9	3
e	001111	2	6
f	101	10	3
g	0010	8	4
h	010	15	3
i	011	17	3
j	00110	4	5

$$WPL=0.19\times 2+(0.15+0.09+0.10+0.15+0.17)\times 3+0.08\times 4+0.04\times 5+(0.02+0.01)\times 6=3.82$$

3.4 树和森林

3.4.1 树与二叉树的转化

根据二叉树的孩子兄弟表示法，可知树和二叉链表一一对应，故可将该二叉链表作为树转化而成的二叉树的存储表示，并称为“左孩子右兄弟”的孩子兄弟存储结构。

1. 转换过程

（1）将结点的第一个孩子结点作为其左孩子，其余各孩子结点为第一个孩子结点的右孩子结点链。

（2）其余结点同样处理。

2. 示例

图 3.6（a）中的树对应的二叉树如图 3.42（b）所示。

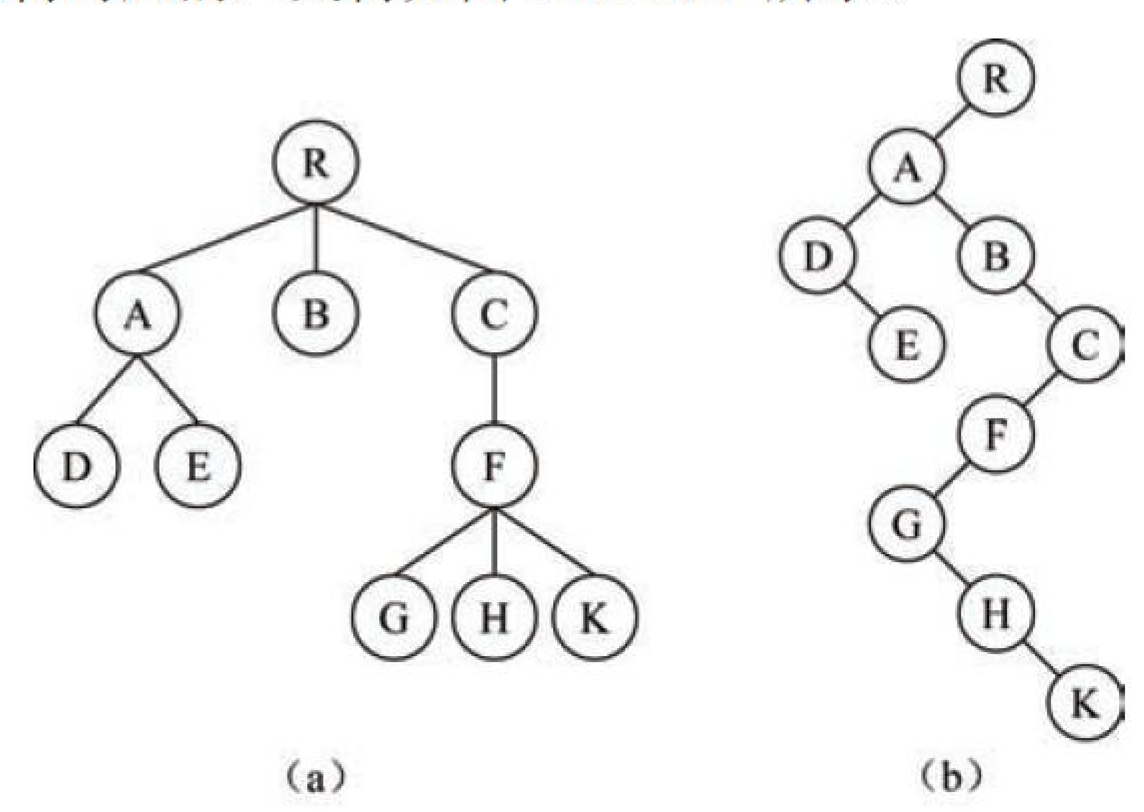


图 3.42　树转换为二叉树示例图

3.4.2 森林与二叉树的转化

森林与二叉树的转化思想和树与二叉树的转化一样。

树的根结点没有兄弟，所以，树所对应的二叉树的根结点的右子树为空。可将森林中其他树的根结点看成第一棵树的根结点的兄弟，可应用树转二叉树的方法求森林转二叉树的方法，如图 3.43 所示，过程如下。

（1）先将每棵树分别转换为对应的二叉树。

（2）将其他二叉树作为第一棵二叉树的右子树。

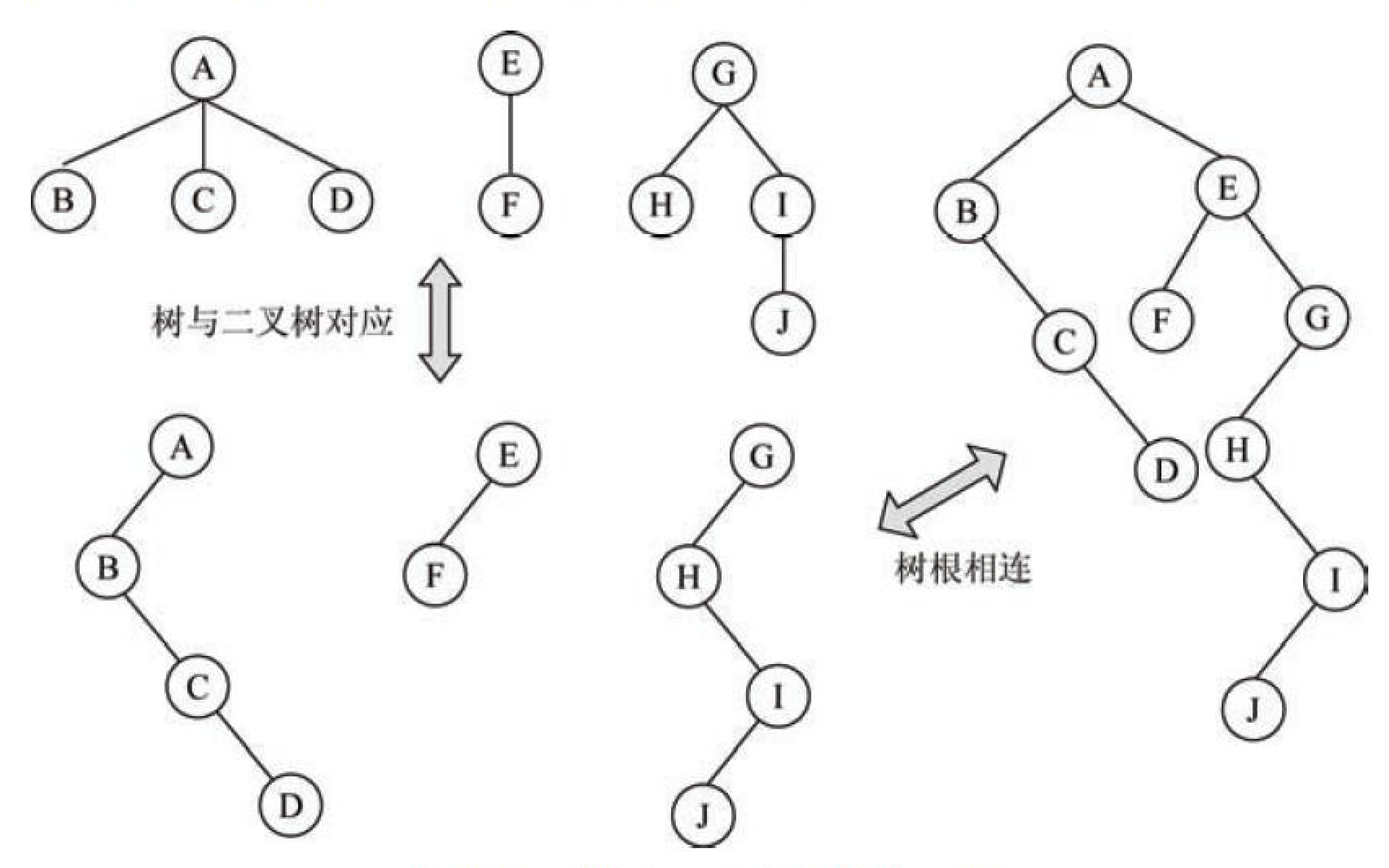


图 3.43 森林与二叉树的转换示例

3.4.3 树的遍历

根据树的特点，树有三种常见的遍历方法。

1）先根遍历

（1）访问树的根结点。

（2）依次先根遍历根的每棵子树。

2）后根遍历

（1）依次后根遍历根结点的每棵子树。

（2）访问根结点。

3）层次遍历

（1）树根入队列。

（2）如果队列不空，则队首元素出队并访问，同时将其孩子结点依次进队列。

（3）重复（2），直至队空。

图 3.44 为树的遍历示例图，其三种遍历序列如下。

（1）先根遍历序列：RADEBCFGHK。

（2）后根遍历序列：DEABGHKFCR。

（3）层次遍历序列：RABCDEFGHK。

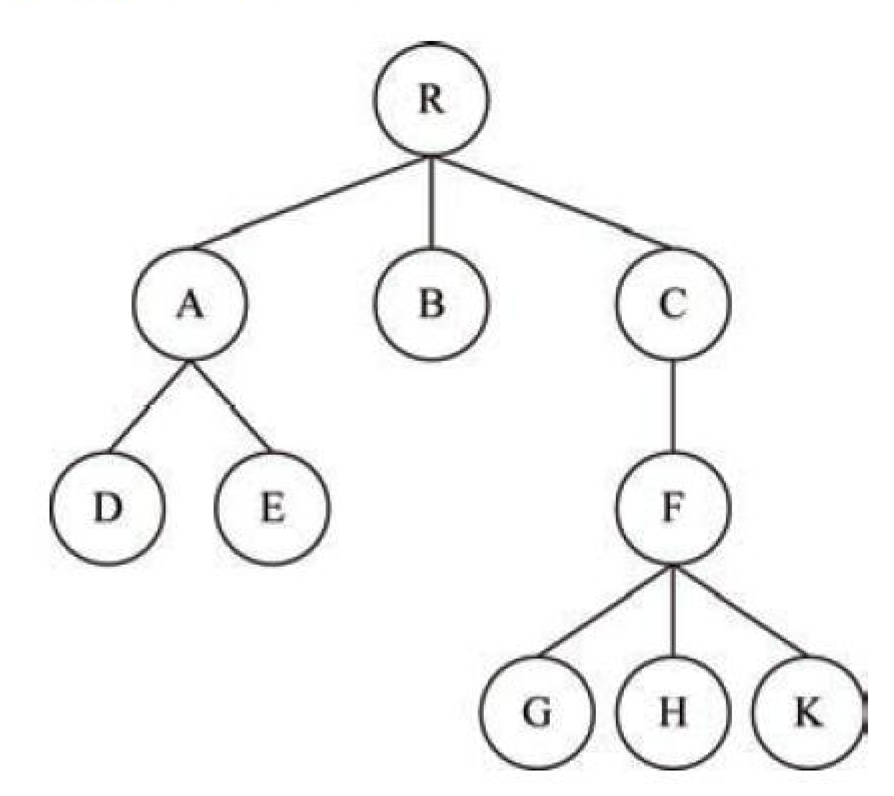


图 3.44　树的遍历序列示例图

3.4.4　森林的遍历

可以将森林分为三部分：第一棵树的根结点；第一棵树的子树森林；其他树构成的森林。由此可得森林的两种遍历方法。

1）先序遍历

（1）若森林不空，则访问森林中第一棵树的根结点。

（2）先序遍历森林中第一棵树的子树森林。

（3）先序遍历森林中除第一棵树之外、其余树构成的森林。

2）后序遍历

（1）若森林不空，则后序遍历森林中第一棵树的子树森林。

（2）访问森林中第一棵树的根结点。

（3）后序遍历森林中除第一棵树之外、其余树构成的森林。

图 3.45 为森林的遍历序列示意图，其两种遍历序列如下。

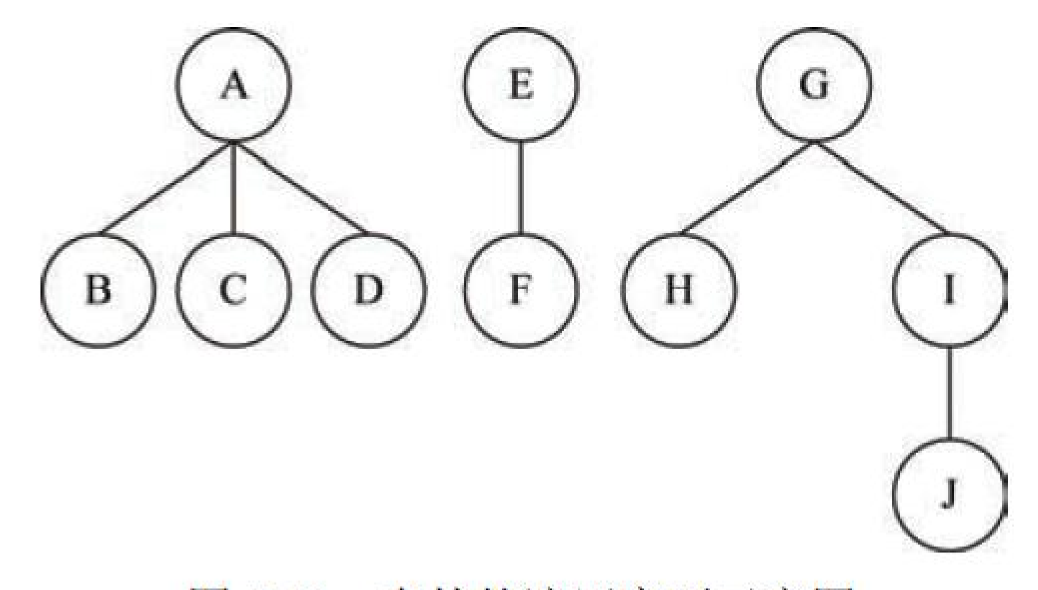


图 3.45　森林的遍历序列示意图

（1）先序序列：ABCDEFGHIJ。

（2）后序序列：BCDAFEHJIG。

结论：树与森林的遍历总结有如下几点。

（1）树的先根遍历对应该树转化二叉树的先序遍历。

（2）树的后根遍历对应该树转化二叉树的中序遍历。

（3）森林的先序遍历对应该森林转化二叉树的先序遍历。

（4）森林的后序遍历对应该森林转化二叉树的中序遍历。

3.5 本章小结

本章内容和其他章节内容存在交集，考点较多。复习过程重点把握基础概念、特征，及算法实现，能够举一反三。另外，非递归算法需要格外注意。

第 4 章　图

本章学习目标

- 理解图的相关概念：有向图、无向图、有向网、无向网。
- 熟练掌握图的存储结构及相关操作、算法实现。
- 深入理解图的遍历思想。
- 理解图的连通性。
- 熟练掌握最小生成树概念及算法。
- 熟练掌握拓扑排序思想及应用。
- 深入理解关键路径、最短路径的含义，熟练掌握相关算法。
- 关注给定存储结构的伪代码实现。

4.1 本章导学



图

4.1.1 知识结构

本章知识结构如图 4.1 所示，加粗框中的内容需要考生重点理解并掌握。

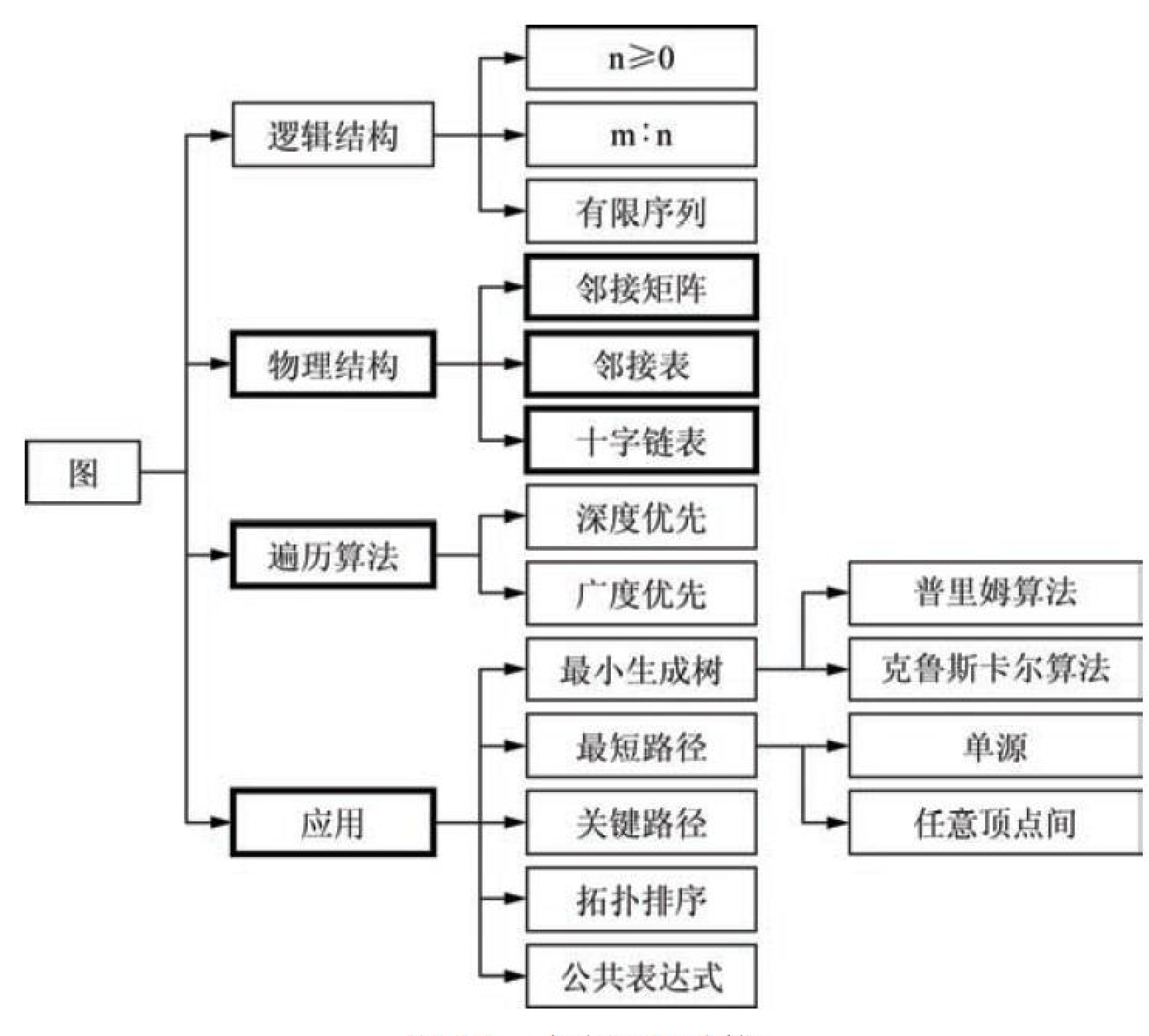


图 4.1　本章知识结构

4.1.2 命题特点

1. 命题规律

（1）本章是历年各高校硕士研究生招生考试的重点考查内容，命题形式多样。

（2）图的相关概念、存储结构、遍历、连通性等易出客观题。

（3）最小生成树、拓扑排序、关键路径、最短路径等客观题、主观题均可出。

2. 命题趋势

本章在全国硕士研究生入学考试“数据结构”部分具有重要地位，尤其需要关注基本概念的内涵、术语、存储结构及相应算法的实现、遍历算法及实现等。应用方面内容较多，如最小生成树，单源、任意顶点间的最短路径，关键路径、拓扑排序、公共表达式等，应深入理解算法内涵、掌握某种存储结构下的算法实现、常见应用等。

4.2 基本概念

图是重要的数据结构，其中各元素的逻辑关系具有 m∶n 的多关联性，每个顶点均可具有多个前驱和后继，定点之间的复杂关系使得图可广泛应用于实际生活中的交通、电信、网络、人物、事务、部门等关系的场合，编程实现时根据实际应用场景可以采用不同的存储结构。

1. 术语

（1）图：是一种比较复杂的非线性结构，其中任意两个元素之间都可以相关。图 G=(V,E)，其中 V 为顶点集，E 为边集，图的特点如下。

① 顶点之间关系任意。

② 顶点之间前驱后继为多对多、m∶n 关系。

③ |V|>0，即顶点数大于 0（有别于线性表和数，图的顶点数不能等于 0）。

④ |E|⩾0，即边数大于或等于 0。

（2）顶点：图中的数据元素。

（3）弧：如果图中顶点的关系用<v,w>表示，则<v,w>表示从顶点 v 到顶点 w 的一条弧，其中 v 是弧尾，w 是弧头。

（4）有向图：顶点的关系用弧表示的图，如图 4.2（a）所示。

（5）无向图：如果图中顶点的关系用（v,w）表示，则（v,w）表示从顶点 v 到顶点 w 的一条边，此时的图称为无向图，如图 4.2（b）所示。

（6）权：和图的边或弧相关的数，该数可以表示相关顶点之间的距离或耗费。

（7）网：带权图。

（8）完全图：结点数为 n，具有 n(n−1)/2 条边的无向图，即任意两个顶点之间均存在一条边。

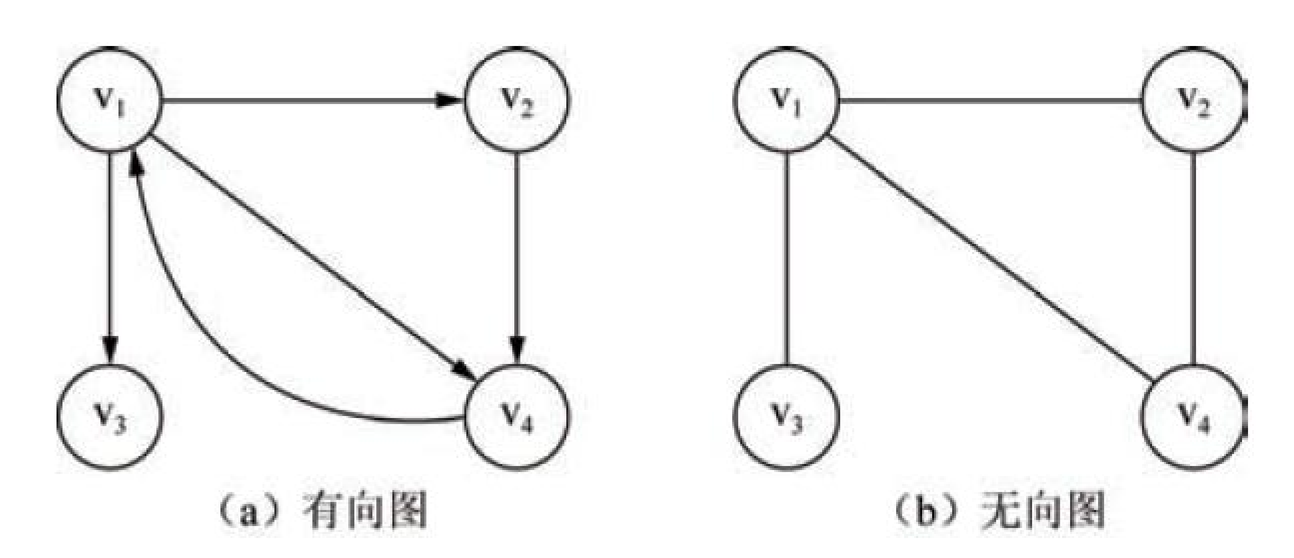


图 4.2　有向图和无向图

（9）有向完全图：结点数为 n，具有 n(n−1)条弧的有向图，即任意两个顶点之间均存在方向相反的两条弧。

（10）稀疏图、稠密图：有很少条边或弧（一般少于 $n\log_2 n$，n 为结点数）的图称为稀疏图，反之称为稠密图。

（11）子图：如果某个图的顶点集为 V，边集或弧集为 E，则由 V 的子集和 E 的子集组合而成的图为原图的子图。

（12）邻接点：在无向图中，如果有边(v,w)，则称顶点 v 和 w 互为邻接点，即 v 和 w 相邻接。边(v,w)依附于顶点 v 和 w，或者说，(v,w)与顶点 v 和 w 相关联。

（13）度：无向图中，顶点 v 的度是指和 v 相关联的边的数目。

（14）入度、出度：有向图中，以顶点 v 为弧头的弧的个数称为顶点 v 的入度，以顶点 v 为弧尾的弧的个数称为顶点 v 的出度。

（15）路径：从顶点 v 到达顶点 w 的一系列边或弧。

① 若路径由弧组成则为有向路径。

② 若路径的起始点和终点相同，则为回路或环。

③ 若路径中不出现相同的顶点，则为简单路径。

④ 若除了第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路，称为简单回路或简单环。

（16）连通：在无向图中，如果从顶点 v 到顶点 w 存在路径，则称顶点 v 和顶点 w 之间连通。

（17）连通图：图的任意两个顶点均连通，无向图中的极大连通子图为其连通分量。

（18）强连通图：任意两个顶点均连通的有向图。

① 有向图的极大强连通子图称为有向图的强连通分量。

② 任何强连通图的强连通分量只有一个，即其本身。

③ 非强连通图有多个强连通分量。

（19）生成树：一个连通图的生成树是一个极小连通子图，含图中全部顶点，但只有足以构成一棵树的 n−1 条边，且边的权值和最小。

2．图的抽象数据类型 ADT 描述

ADT graph{
　　数据对象 V：V={ a_i| a_i|∈eleSet， i=1,2,⋯,n， n>0}，称 V 为顶点集

数据关系 E：E={< a_i，a_j > | a_i，a_j∈V,且 p(a_i，a_j)，i,j∈[1,n]，< a_i，a_j > 表示从 a_i 到 a_j 的边，p(a_i，a_j)定义边< a_i，a_j >的含义}

基本操作 P：

```
    void createGraph(&G,V,E)          // 初始化图 G，V 为顶点集，E 为边集
    void destroyGraph(&G)             // 销毁图 G
    unsigned  locateVex(G, u)         // 获取顶点 u 在图 G 中的位置
    unsigned  GetVex(G, v)            // 获取图 G 的顶点 v
    int  firstAdjVex(G, v)            // 获取 v 在 G 中的第一个邻接点
    int  nextAdjVex(G, v, w)          // 获取 v 在 G 中邻接点 w 之后的邻接点
    void insertVex(&G, v)             // 图 G 中插入顶点 v
    void deleVex(&G, v)               // 删除图 G 中的顶点 v
    void insertEdge(&G, v, w)         // 图 G 中插入边<v,w>
    void deleEdge(&G, v, w)           // 删除图 G 中的边<v,w>
    int  getEdgeValue(G, v, w)        // 获取图 G 中的边<v,w>的权值
    void setEdgeValue(G, v, w,x)      // 设置图 G 中边<v,w>的权值为 x
    void DFSTraverse(G, Visit( ) )    // 深度优先 Visit 图 G
    void BFSTraverse(G, Visit( ) )    // 宽度优先 Visit 图 G
    void miniTreeOfGraph(G,&T)        // 求 G 的最小生成树 T
    void topSortGraph(G,&TS)          // 求 G 的拓扑序列 TS
    void miniPathOfGraph(G,&p)        // 求 G 的最短路径 p
    void keyPathOfGraph(G,&p)         // 求 G 的关键路径 p
} ADT graph
```

4.3 存储结构

存储结构有邻接矩阵、邻接表和十字链表三种。

4.3.1 邻接矩阵

1. 数组表示法

设图 G=(V,E)，n=|V|表示顶点个数，m=|E|表示边的个数。图的邻接矩阵存储结构特点如下。

（1）用一个 n×n 的二维数组 a[n][n]表示图的边或弧的信息。

- 存储空间仅和顶点数有关，适用于稠密图的存储。
- 需 n^2 个数组元素的存储空间。

（2）用一个一维数组表示图的结点信息。

（3）直观、易于编程实现，适用于稠密图。对于稀疏图浪费存储空间，且顶点的插入删除等操作实现麻烦。

无向图的邻接矩阵特点如下。

（1）a[i][j]表示顶点 v[i]到 v[j]之间是否有边（无向图）或边的权值（无向网），有边则 a[i][j]≠0，否则 a[i][j]=0，i, j∈[0,n−1]。

（2）a[i][i]=0。

（3）a[i][j]=a[j][i]。

（4）可压缩存储，仅需 n(n-1)/2 个数组元素，存储上或下三角阵的除对角线以外元素。

（5）第 i 行中非零元素个数为顶点 v[i]的度。

有向图的邻接矩阵特点如下。

（1）a[i][j]表示顶点 v[i]到 v[j]之间是否有弧（有向图）或弧长（有向网），无弧则 a[i][j]为∞，i, j∈[0,n-1]。

（2）a[i][i]=∞。

（3）第 i 行中非零元素个数为顶点 v[i]的出度；第 i 列中非零元素个数为顶点 v[i]的入度。

2．邻接矩阵图表示

邻接矩阵图如图 4.3 所示。

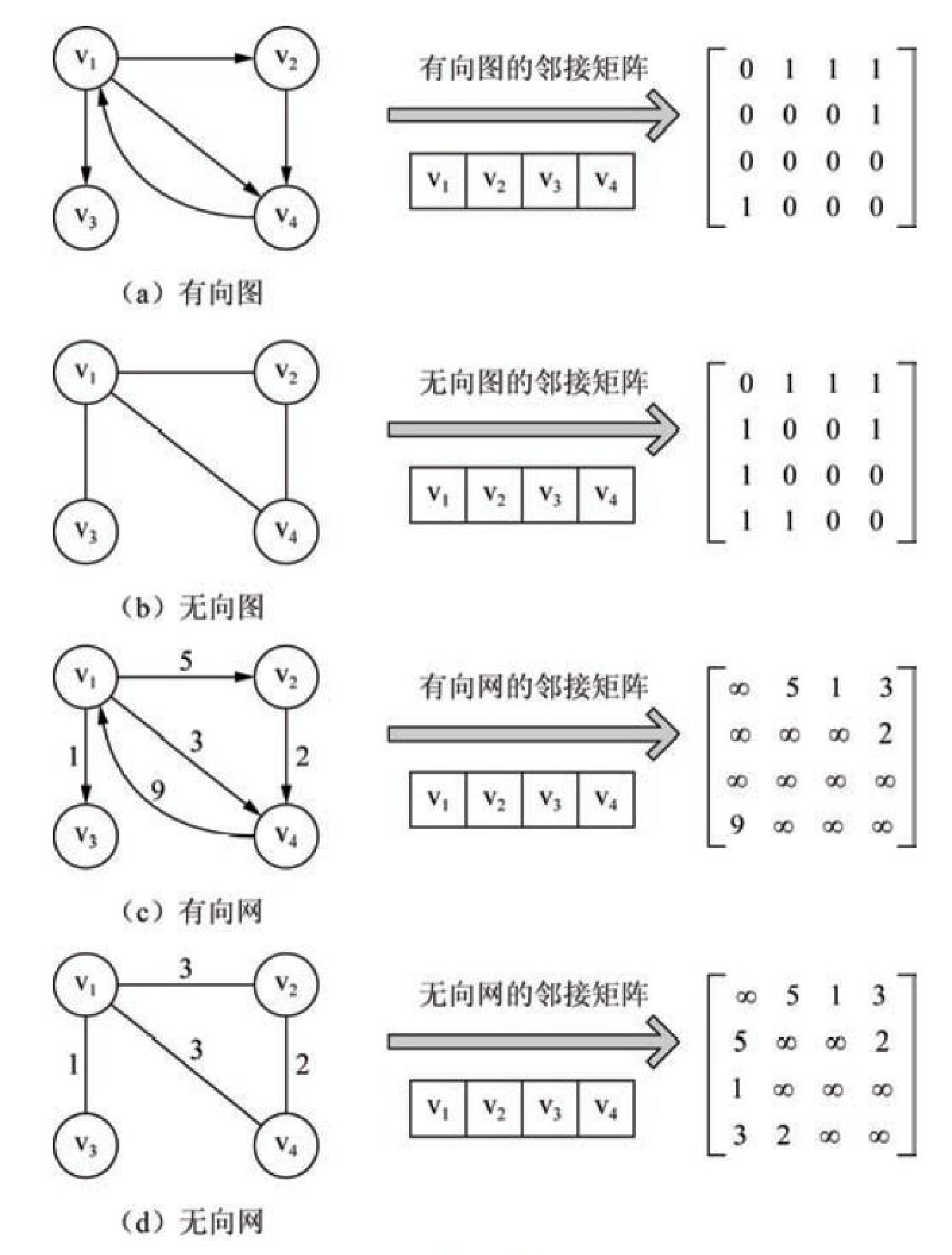


图 4.3　图的邻接矩阵存储结构

3．定义

```
#define INFINITY   65535            // 定义最大值，16 位字长最大 65535，相当于∞
#define MAX_VEX_NUM  100            // 定义最多顶点个数
```

```
typedef enum {DG, DN, UDG, UDN} graphKind;
                                  // 定义图的类型：有向图，有向网，无向图，无向网
typedef char vertexType;          // 定义顶点类型为字符型
typedef struct{                   // 定义邻接矩阵
    VRType adj;                   // VRType 是顶点关系的类型，一般可用 int
                                  // 无权图用 1 或 0 表示是否相邻，带权图为权值类型
    InfoType *info;               // 弧相关信息的指针，比如权值，可以无此项
} arcNode, adjMatrix[MAX_VEX_NUM][MAX_VEX_NUM];
typedef struct {                  // 定义邻接矩阵表示的图类型 matrixGraph
    vertexType vexs[MAX_VEX_NUM]; // 顶点向量
    adjMatrix arcs;               // 邻接矩阵
    int vexNum, arcNum;           // 图的顶点数和弧(边)数
    graphKind kind;               // 图的种类标志
} matrixGraph;
```

4. 操作实现举例

1）创建图

（1）创建无向图。

```
void createUDG(matrixGraph  &G) {
    G.kind=UDG;                               // 设置图的类型为无向图 UDG
    scanf( "%d%d" ,&G.vexNum,&G.arcNum);      // 输入无向图的顶点数和边数
    for(i=0;i<G.vexNum;i++)                   // 输入顶点信息
        scanf("%c",&G.vexs[i]);
    for(i=0;i<G.vexNum;i++)
        for(j=0;j<G.vexNum;j++)
            G.arcs[i][j]=0;                   // 所有边赋初值 0
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d" ,&i,&j);                 // 读入边信息，i，j 是边的顶点序号
        G.arcs[i][j]=G.arcs[j][i]=1;
    }
}
```

（2）创建无向网。

```
void createUDN(matrixGraph  &G) {
    G.kind=UDN;                               // 设置图的类型为无向网 UDN
    scanf("%d%d",&G.vexNum,&G.arcNum);        // 输入无向网的顶点数和边数
    for(i=0;i<G.vexNum;i++)                   // 输入顶点信息
        scanf("%c",&G.vexs[i]);
    for(i=0;i<G.vexNum;i++)
        for(j=0;j<G.vexNum;j++)
            G.arcs[i][j]=INFINITY;            // 所有边的权值赋初值∞
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d%d" ,&i,&j,&w);            // 读入边信息，i，j 是边的顶点序号
        G.arcs[i][j]=G.arcs[j][i]=w;
    }
}
```

（3）创建有向图。

```
void createDG(matrixGraph  &G) {
```

```
    G.kind=DG;                                  // 设置图的类型为有向图 DG
    scanf("%d%d",&G.vexNum,&G.arcNum);          // 输入有向图的顶点数和边数
    for(i=0;i<G.vexNum;i++)                     // 输入顶点信息
        scanf("%c",&G.vexs[i]);
    for(i=0;i<G.vexNum;i++)
        for(j=0;j<G.vexNum;j++)
            G.arcs[i][j]=INFINITY;              // 所有边赋初值∞
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d",&i,&j);                    // 读入边信息，i，j 是边的顶点序号
        G.arcs[i][j]=1;
    }
}
```

（4）创建有向网。

```
void createDN(matrixGraph  &G) {
    G.kind=DN;                                  // 设置图的类型为有向网 DN
    scanf("%d%d",&G.vexNum,&G.arcNum);          // 输入有向网的顶点数和边数
    for(i=0;i<G.vexNum;i++)                     // 输入顶点信息
        scanf("%c" ,&G.vexs[i]);
    for(i=0;i<G.vexNum;i++)
        for(j=0;j<G.vexNum;j++)
            G.arcs[i][j]=INFINITY;              // 所有边赋初值∞
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d%d",&i,&j,&w);               // 读入边信息，i，j 是边的顶点序号
        G.arcs[i][j]=w;
    }
}
```

注意：

- 采用邻接矩阵的算法时间复杂度为 $O(n^2)$，和边数无关，所以适合作为稠密图的存储结构。
- 可在算法中读入图的类型，并根据输入的类型确定邻接矩阵对应二维数组的赋值，从而将上述四个算法合并为一个。请考生自行修改完成。

2）插入边

```
void insertEdge(matrixGraph  &G, vertexType  v, vertexType w){
                              // 在顶点 v 和 w 之间插入一条边
    i=locateVex(G, v);        // 获取顶点 v 在无向图 G 中的位置
    j=locateVex(G, w);        // 获取顶点 w 在无向图 G 中的位置
    printf("Please input the type of Graph(0:DG, 1:DN, 2:UDG, 3:UDN)\n");
    scanf("%d",&k);
    switch(k){
        case 0:
        case 2: G.arcs[i][j]=1; break;
        case 1:
        case 3: scanf("%d",&ew); G.arcs[i][j]=ew;
    }
}
```

其他基本操作请考生自行练习。

4.3.2 邻接表

1．链式存储结构

链式存储结构的特点如下。

（1）用一个一维数组表示图的顶点信息，每个元素包含 2 个域：顶点 v_i 及指向该顶点相关边的单链表指针。

（2）每个顶点建立一个单链表，表示和该顶点有关的边或弧的信息。

① 每个单链表附设一个表头结点，在表头结点中，除了设有链域（firstArc）指向链表中第一个结点之外，还设有存储顶点 v_i 相关信息的数据域。该表头结点即为（1）中的数组元素。

② 第 i 个单链表中的结点表示依附于顶点 v_i 的边（对于有向图是以顶点 v_i 为弧尾的弧，如果是逆邻接表的话，则是以顶点 v_i 为弧头的弧）。

③ 每个单链表结点由 3 个域组成：

- 邻接点域（adjVex）：指示与顶点 v_i 邻接的点在图中的位置。
- 链域（nextArc）：指示下一条边或弧的结点。
- 数据域（info）：存储和边或弧相关的信息，如权值等，一般会省略。

④ 邻接表中每个顶点的单链表中结点个数为该顶点的出度。

⑤ 逆邻接表中每个顶点的单链表中结点个数为该顶点的入度。

（3）存储空间主要由边数决定，适合稀疏图，插入删除操作相对简单。

2．图的邻接表表示

图的邻接表如图 4.4 所示。

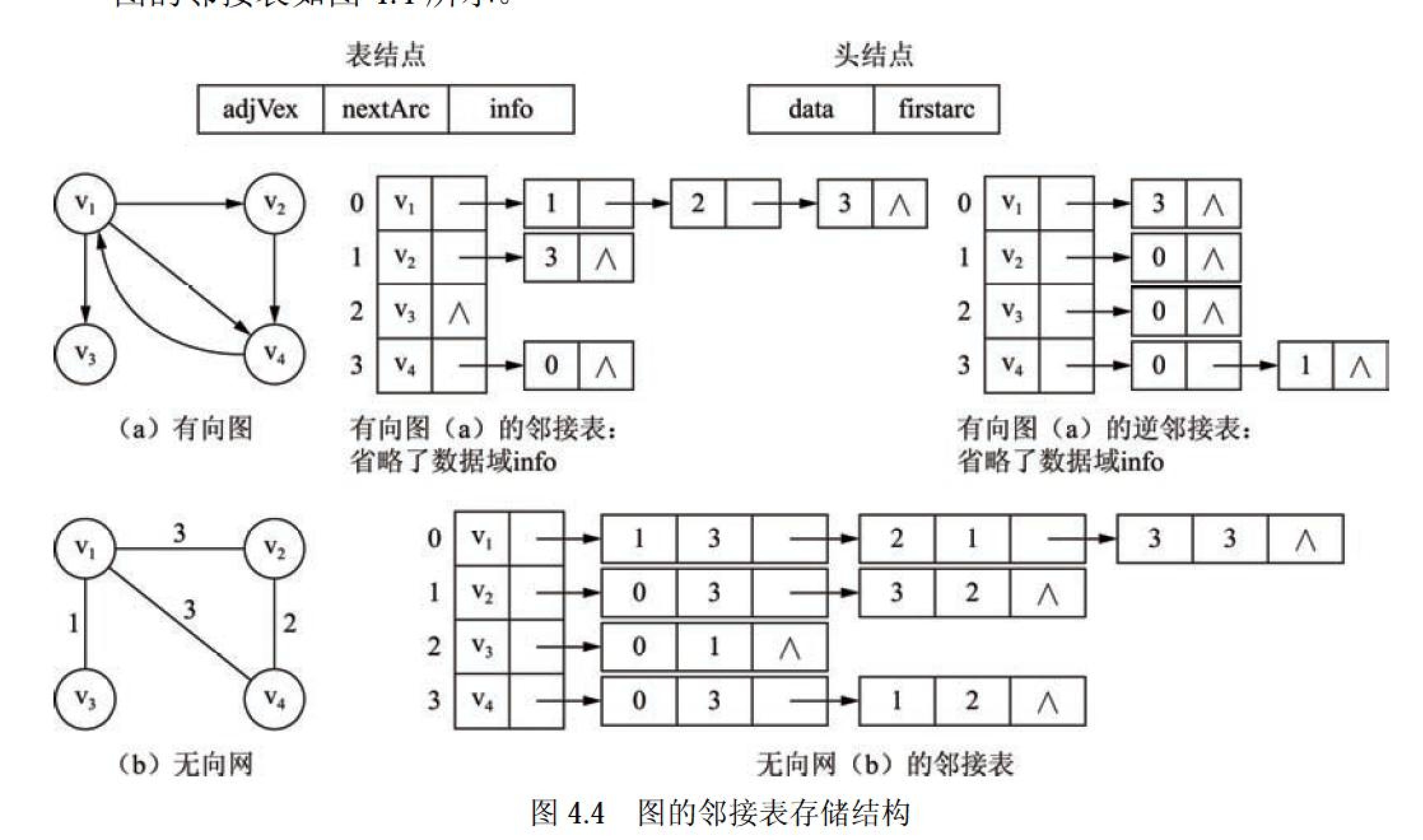


图 4.4　图的邻接表存储结构

3. 邻接表的定义

```
#define  MAX_VEX_NUM   100         // 定义最多顶点个数
typedef enum {DG, DN, UDG, UDN} graphKind;
                                   // 定义图的类型：有向图，有向网，无向图，无向网
typedef char vertexType;           // 定义顶点类型为字符型
typedef struct  tANode {
    int   adjVex;                  // 该弧所指向的顶点的位置
    struct  tANode   *nextArc;     // 指向下一条弧的指针
    InfoType   *info;              // 该弧相关信息指针，比如权值，可无此项
} tAarcNode;
typedef struct {                   // 定义一维数组存储顶点和其边相关的单链表信息
    vertexType  data;              // 顶点
    tAarcNode  *firstArc;          // 指向第一条依附该顶点的弧
} TNode, adjList[MAX_VEX_NUM];
typedef struct {                   // 定义邻接表
    adjList   vertices;
    int   vexNum, arcNum;          // 图的当前顶点数和弧数
    int   kind;                    // 图的种类标志
} adjListGraph
```

4. 操作实现举例

1）创建图

（1）创建无向图。

```
void createUDG(adjListGraph  &G) {
    G.kind=UDG;                               // 设置图的类型为无向图 UDG
    scanf("%d%d",&G.vexNum,&G.arcNum);        // 输入无向图的顶点数和边数
    for(i=0;i<G.vexNum;i++){                  // 初始化一维数组的顶点相关信息
        scanf("%c",&G.vertices[i].data);
        G.vertices[i].firstArc=NULL;
    }
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d",&i,&j);                  // 读入边信息，i，j 是边的顶点序号
        p=( tAarcNode *)malloc(sizeof(tAarcNode));
        p->adjVex=j;
        p->nextArc = G.vertices[i].firstArc;
        G.vertices[i].firstArc=p;
        p=( tAarcNode *)malloc(sizeof(tAarcNode));
        p->adjVex=i;
        p->nextArc = G.vertices[j].firstArc;
        G.vertices[j].firstArc=p;
    }
}
```

（2）创建无向网。

```
void createUDN(adjListGraph  &G) {
    G.kind=UDN;                               // 设置图的类型为无向网 UDN
    scanf("%d%d",&G.vexNum,&G.arcNum);        // 输入无向网的顶点数和边数
    for(i=0;i<G.vexNum;i++){                  // 初始化一维数组的顶点相关信息
        scanf("%c",&G.vertices[i].data);
        G.vertices[i].firstArc=NULL;
```

```
    }
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d%d",&i,&j,&w);          // 读入边信息，i，j 是边的顶点序号
        p=( tAarcNode *)malloc(sizeof(tAarcNode));
        p->adjVex=j;
        p->info=w;
        p->nextArc = G.vertices[i].firstArc;
        G.vertices[i].firstArc=p;
        p=( tAarcNode *)malloc(sizeof(tAarcNode));
        p->adjVex=i;
        p->info=w;
        p->nextArc = G.vertices[j].firstArc;
        G.vertices[j].firstArc=p;
    }
}
```

（3）创建有向图。

```
void createDG(adjListGraph  &G) {
    G.kind=DG;                              // 设置图的类型为有向图 DG
    scanf("%d%d",&G.vexNum,&G.arcNum);      // 输入有向图的顶点数和边数
    for(i=0;i<G.vexNum;i++){                // 初始化一维数组的顶点相关信息
        scanf("%c",&G.vertices[i].data);
        G.vertices[i].firstArc=NULL;
    }
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d",&i,&j);                // 读入边信息，i，j 是边的顶点序号
        p=( tAarcNode *)malloc(sizeof(tAarcNode));
        p->adjVex=j;
        p->nextArc = G.vertices[i].firstArc;
        G.vertices[i].firstArc=p;
    }
}
```

（4）创建有向网。

```
void createDN(adjListGraph  &G) {
    G.kind=DN;                              // 设置图的类型为有向网 DN
    scanf("%d%d",&G.vexNum,&G.arcNum);      // 输入有向网的顶点数和边数
    for(i=0;i<G.vexNum;i++){                // 初始化一维数组的顶点相关信息
        scanf("%c",&G.vertices[i].data);
        G.vertices[i].firstArc=NULL;
    }
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d%d",&i,&j,&w);          // 读入边信息，i，j 是边的顶点序号
        p=( tAarcNode *)malloc(sizeof(tAarcNode));
        p->adjVex=j;
        p->info=w;
        p->nextArc = G.vertices[i].firstArc;
        G.vertices[i].firstArc=p;
    }
}
```

注意：

- 采用邻接表的算法时间复杂度为 O(n+e)，和顶点数及边数有关，适合作为边数较少的图的存储结构。
- 可在算法中读入图的类型，并根据输入的类型确定邻接表中结点的赋值，从而将上述四个算法合并为一个。请考生自行修改完成。

2）插入边

```
void insertEdge(adjListGraph &G, vertexType v, vertexType w){
                                            // 在顶点 v 和 w 之间插入一条边
    i=locateVex(G, v);                      // 获取顶点 v 在无向图 G 中的位置
    j=locateVex(G, w);                      // 获取顶点 w 在无向图 G 中的位置
    p=( tAarcNode *)malloc(sizeof(tAarcNode));
    p->adjVex=j;
    p->nextArc = G.vertices[i].firstArc;
    G.vertices[i].firstArc=p;
    printf("Please input the type of Graph(1:DN, 2:UDG, 3:UDN)\n");
    scanf("%d",&k);
    switch(k){
        case 1: scanf("%d",&w); p->info=w; break;
        case 2: p=( tAarcNode *)malloc(sizeof(tAarcNode));
              p->adjVex=i; p->nextArc = G.vertices[j].firstArc;
              G.vertices[j].firstArc=p;break;
        case 3: scanf("%d",&w); p->info=w;
              p=( tAarcNode *)malloc(sizeof(tAarcNode));
              p->adjVex=i; p->nextArc = G.vertices[j].firstArc;
              G.vertices[j].firstArc=p;
    }
}
```

其他基本操作请考生自行练习。

4.3.3　十字链表

1．特点

十字链表是有向图的链式存储结构，特点如下。

（1）用一个一维数组表示图的顶点信息，每个元素包含 3 个域：顶点 v_i、数据域 data，指向第一条以该顶点为头的边或弧指针 firstin，第一条以该顶点为尾的边或弧指针 firstout。

（2）边或弧信息包含该边对应两个顶点的位置信息 headVex 和 tailVex，下一个同头边或同头弧指针 hlink，下一个同尾边或同尾弧指针 tlink，以及和该边或弧相关的其他信息指针 info。

（3）hlink 链的结点个数为表头顶点的入度，tlink 链的结点个数为表头顶点的出度。

（4）是有向图的邻接表和逆邻接表存储结构的结合。

2．图的十字链表表示

图的十字链表如图 4.5 所示。

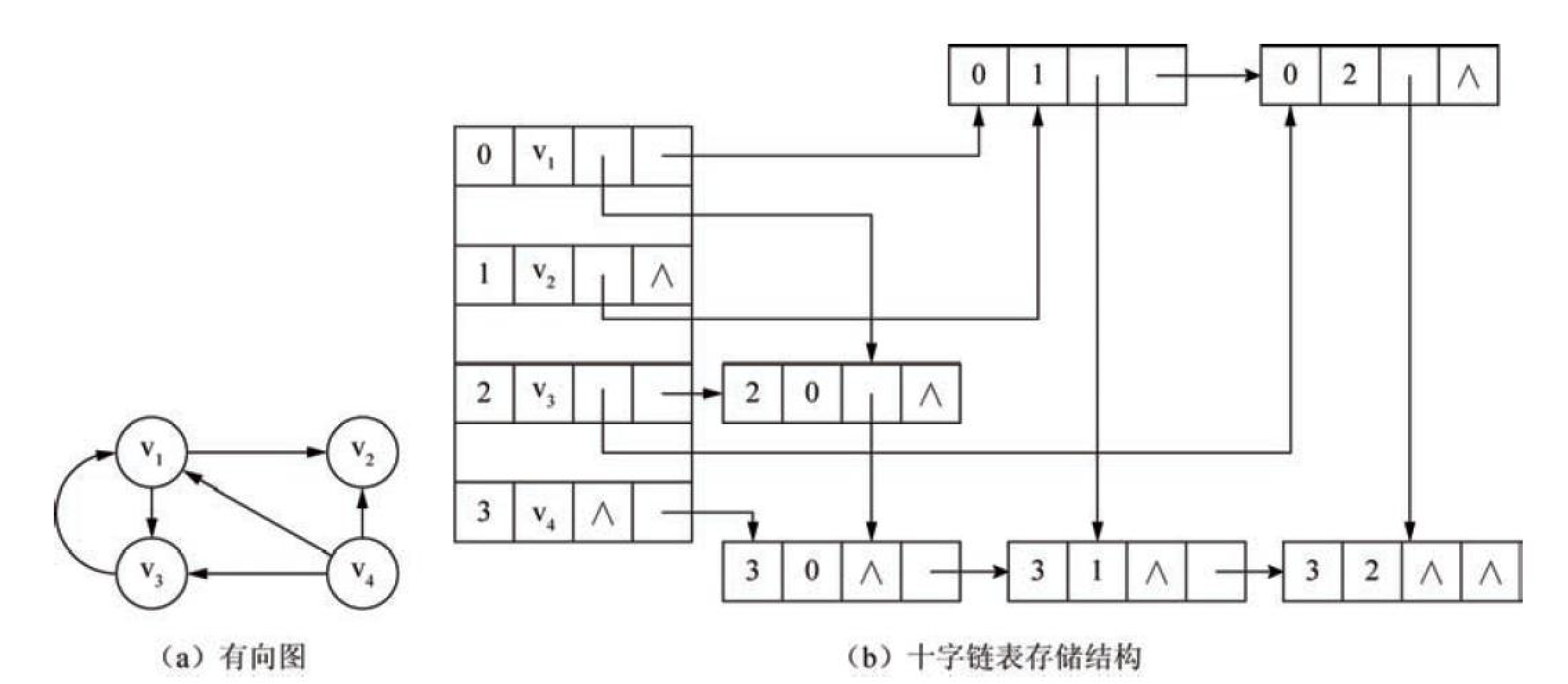


图 4.5　有向图的十字链表存储

3. 定义

```
#define  MAX_VEX_NUM   100                  // 定义最多顶点个数
typedef enum {DG, DN } cGraphKind;          // 定义图的类型：有向图，有向网
typedef char vertexType;                    // 定义顶点类型为字符型
typedef struct aBox {
   int  headVex, tailVex;                   // 该弧的头和尾顶点位置
   struct  aBox  *hlink,  *tlink;
       // hlink 和 tlink 分别指向下一个弧头相同和弧尾相同的弧的指针域
   InfoType *info;                          // 该弧相关信息，比如权值，可无此项
} arcBox;
typedef struct {
   vertexType  data;
   arcBox   *firstin,  *firstout;           // 分别指向该顶点第一条入弧和出弧
} vexNode;
typedef struct {
   vexNode  xlist[MAX_VEX_NUM];             // 表头数组
   int   vexNum, arcNum;                    // 有向图的顶点数和弧数
cGraphKind  kind;
} crossLinkGraph;
```

4. 操作实现举例

1）创建图

（1）创建有向图。

```
void createDG(crossLinkGraph  &G) {
    G.kind=DG;                              // 设置图的类型为有向图 DG
    scanf("%d%d",&G.vexNum,&G.arcNum);      // 输入有向图的顶点数和边数
    for(i=0;i<G.vexNum;i++){                // 初始化一维数组的顶点相关信息
        scanf("%c",&G. xlist[i].data);
        G.xlist[i].firstin=NULL;
        G.xlist[i].firstout=NULL;
    }
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d",&i,&j);                // 读入边信息，i，j 是边的顶点序号
```

```
        p=( arcBox *)malloc(sizeof(arcBox));
        p->tailVex =i;
        p->headVex =j;
        p->hlink = G.xlist[j].firstin;
        p->tlink = G.xlist[i].firstout;
        G.xlist[j].firstin=p;
        G.xlist[i].firstout=p;
    }
}
```

（2）创建有向网。

```
void createDG(crossLinkGraph  &G) {
    G.kind=DG;                                // 设置图的类型为有向图 DG
    scanf("%d%d",&G.vexNum,&G.arcNum);        // 输入有向图的顶点数和边数
    for(i=0;i<G.vexNum;i++){                  // 初始化一维数组的顶点相关信息
        scanf("%c",&G. xlist[i].data);
        G.xlist[i].firstin=NULL;
        G.xlist[i].firstout=NULL;
    }
    for(k=0;k<G.arcNum;k++)
    {
        scanf("%d%d%d",&i,&j,&w);             // 读入边信息，i，j 是边的顶点序号
        p=( arcBox *)malloc(sizeof(arcBox));
        p->tailVex =i;
        p->headVex =j;
        p->info =w;
        p->hlink = G.xlist[j].firstin;
        p->tlink = G.xlist[i].firstout;
        G.xlist[j].firstin=p;
        G.xlist[i].firstout=p;
    }
}
```

2）插入边

```
void insertEdge(crossLinkGraph  &G, vertexType  v, vertexType  w){
                                              // 在顶点 v 和 w 之间插入一条边
    i=locateVex(G, v);                        // 获取顶点 v 在无向图 G 中的位置
    j=locateVex(G, w);                        // 获取顶点 w 在无向图 G 中的位置
    p=( arcBox *)malloc(sizeof(arcBox));
    p->tailVex =i;
    p->headVex =j;
    p->hlink = G.xlist[j].firstin;
    p->tlink = G.xlist[i].firstout;
    G.xlist[j].firstin=p;
    G.xlist[i].firstout=p;
    printf("Please input the type of Graph(1:UDG, 2:DN)\n");
    scanf("%d",&k);
    if(k==2)
        scanf("%d",&w); p->info =w;
}
```

其他基本操作请考生自行练习。

4.4 遍　　历

图的遍历和树的遍历类似，从图中某一顶点出发遍访图中其余顶点，且使每一个顶点仅被访问一次。连通图的遍历产生边数最少的生成树，该算法是图应用的基础。

4.4.1 深度优先搜索

深度优先遍历类似于树的先根遍历，是树的先根遍历的推广。

假设图中所有顶点均未被访问，则深度优先搜索过程如下。

（1）从图中某顶点 v 出发，访问此顶点。

（2）依次从 v 的未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点均被访问。

（3）若此时图中尚有顶点未被访问，则选择图中一个未曾被访问的顶点作起始点，重复上述过程，直到图中所有顶点都被访问为止。

注意：

- 如果仅给出图的逻辑结构(没给具体存储结构)，则其深度优先遍历序列不唯一。
- 存储结构一旦确定并给出，遍历序列唯一。

深度优先遍历算法的详细描述如下。

1. 邻接矩阵存储结构

1）算法描述

```
Boolean visited[MAX_VEX_NUM];        // 是否访问标志数组
#define  TRUE   1
#define  FALSE  0
void DFSTraverse(matrixGraph  G) {
   for (i=0; i<G.vexNum; i++)
      visited[i]=FALSE;              // 访问标志数组初始化
   for (i=0; i<G.vexNum; i++)
      if (!visited[i])
         DFS(G, i);                  // 对尚未访问的顶点调用 DFS
}
void DFS(matrixGraph, int i) {       // 从第 i 个顶点出发深度优先搜索图 G
   visite(G.vexs[i]);
   visited[i]=TRUE;
   for(j=0;j< G.vexNum;j++){
      if (!visited[j] && (G.arcs[i][j]!=0 || G.arcs[i][j]!=INFINITY))
         DFS(G, j);
}
```

2）示例

某有向图如图 4.6（a）所示，图 4.6（b）为其邻接矩阵，从 v_1 开始的深度优先遍历过程如图 4.7 所示。

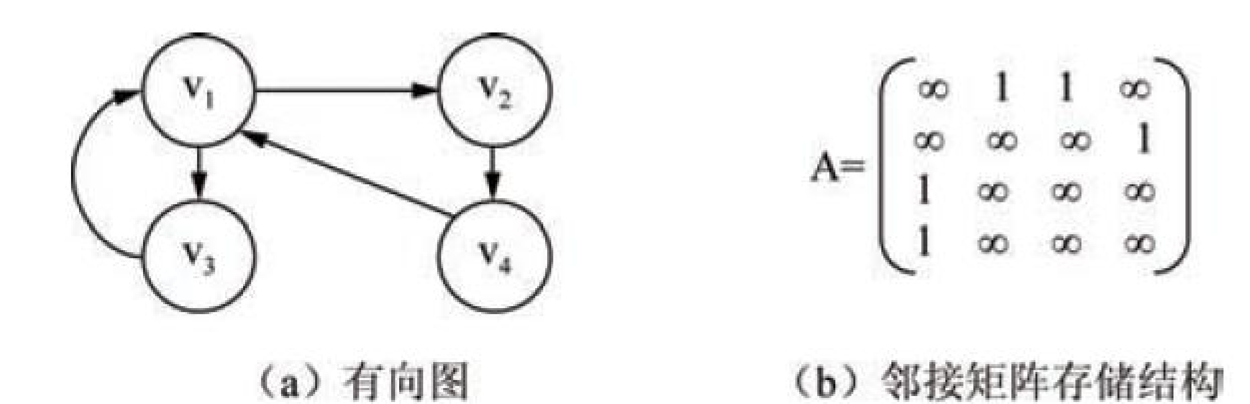


图 4.6　有向图的邻接矩阵存储

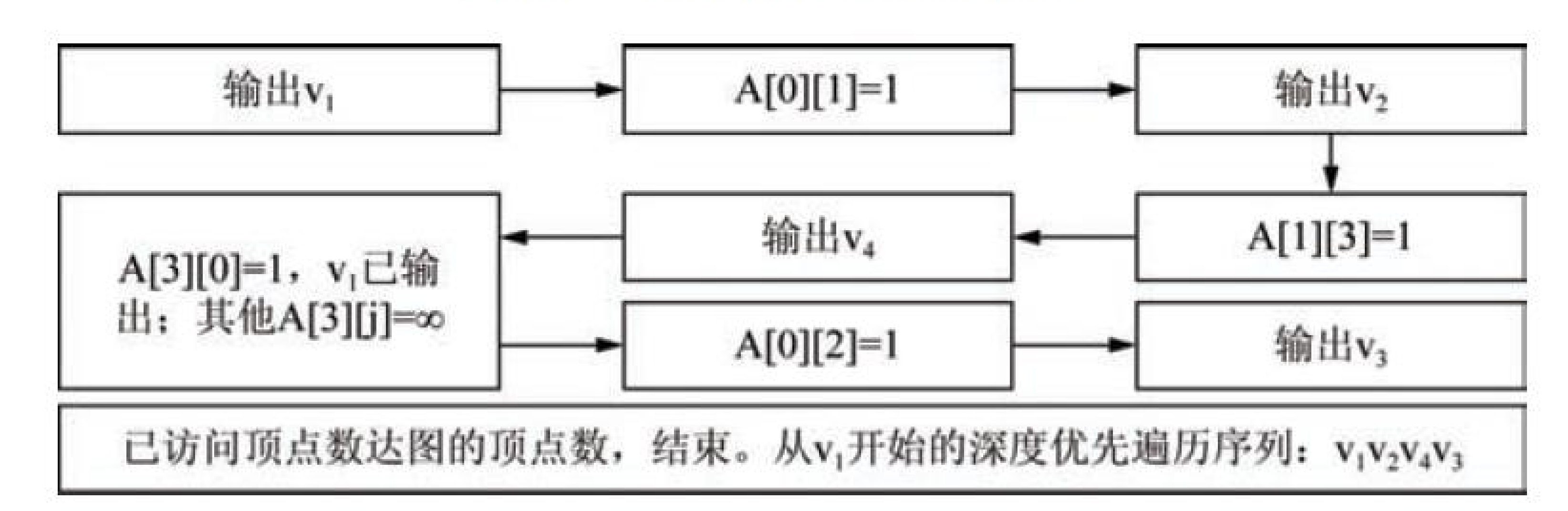


图 4.7　有向图的邻接矩阵存储深度优先遍历过程

2．邻接表存储结构

1）算法描述

```
Boolean visited[MAX_VEX_NUM];          // 是否访问标志数组
#define  TRUE   1
#define  FALSE  0
void DFSTraverse(adjListGraph  G) {
    for  (i=0;  i<G.vexNum;  i++)
        visited[i]=FALSE;              // 访问标志数组初始化
    for  (i=0;  i<G.vexNum;  i++)
        if  (!visited[i])
            DFS(G,  i);                // 对尚未访问的顶点调用 DFS
}
void DFS(adjListGraph,  int  i)  {     // 从第 i 个顶点出发深度优先搜索图 G
    visite(G. vertices[i].data);
    visited[i]=TRUE;
    p=G.vertices[i].firstArc;
    for(;p!=NULL,j=p->adjVex;p=p->nextArc)
        if  (!visited[j])
            DFS(G,  j);
}
```

2）示例

某有向图如图 4.8（a）所示，图 4.8（b）为其邻接矩阵，从 v_1 开始的深度优先遍历过程如图 4.9 所示。

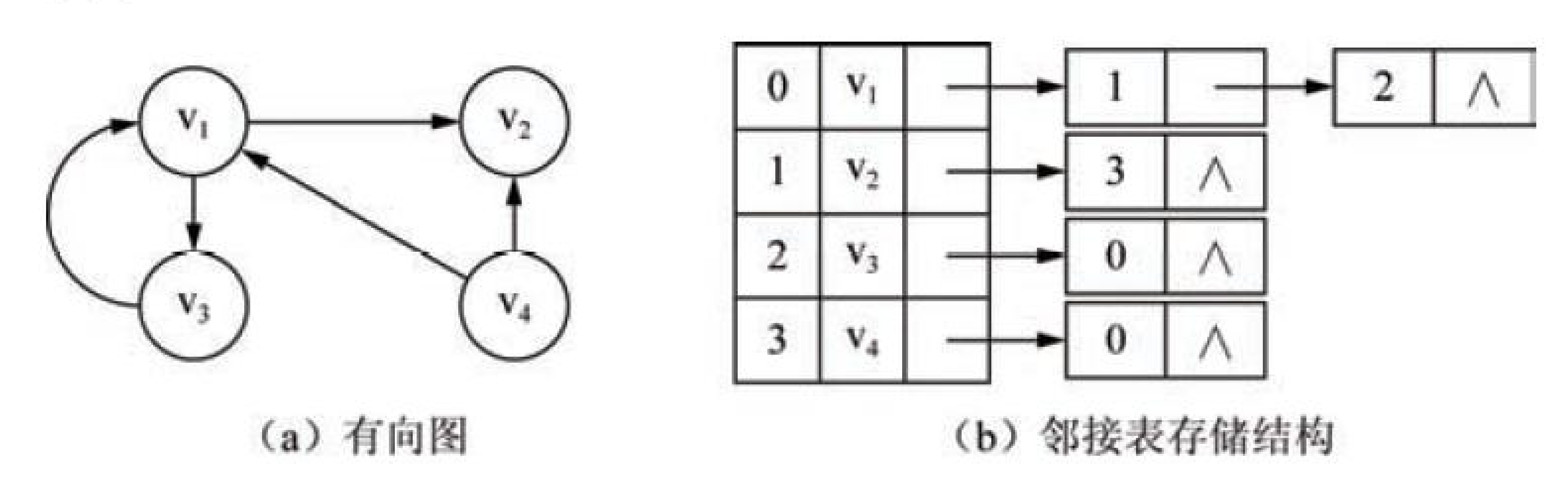


图 4.8　有向图的邻接表存储

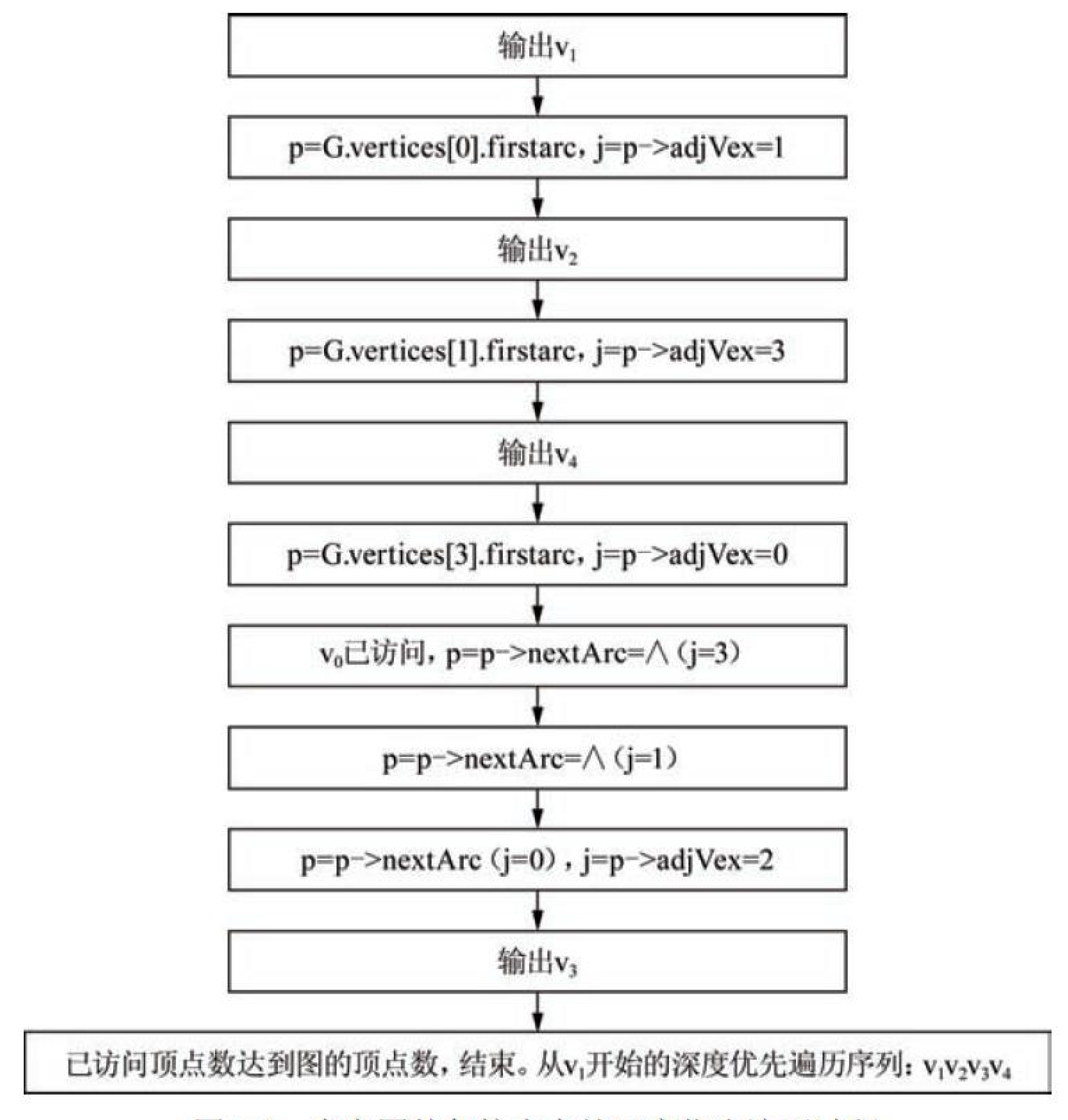


图 4.9　有向图的邻接表存储深度优先遍历过程

3．十字链表存储结构

1）算法描述

```
Boolean visited[MAX_VEX_NUM];          // 是否访问标志数组
#define  TRUE   1
#define  FALSE  0
void DFSTraverse(crossLinkGraph  G) {
   for  (i=0; i<G.vexNum; i++)
      visited[i]=FALSE;                // 访问标志数组初始化
   for  (i=0; i<G.vexNum; i++)
      if  (!visited[i])
         DFS(G, i);                    // 对尚未访问的顶点调用 DFS
}
void DFS(crossLinkGraph, int i) {      // 从第 i 个顶点出发深度优先搜索图 G
visite(G.xlist[i].data);
visited[i]=TRUE;
p= G.xlist[i].fistout;
for (;p!=NULL,j=p->headVex;p=p->tlink){
          if (!visited[j])
      DFS(G, j);
}
```

2）示例

某有向图如图 4.10（a）所示，图 4.10（b）为其邻接矩阵，从 v_1 开始的深度优先遍历过程如图 4.11 所示。

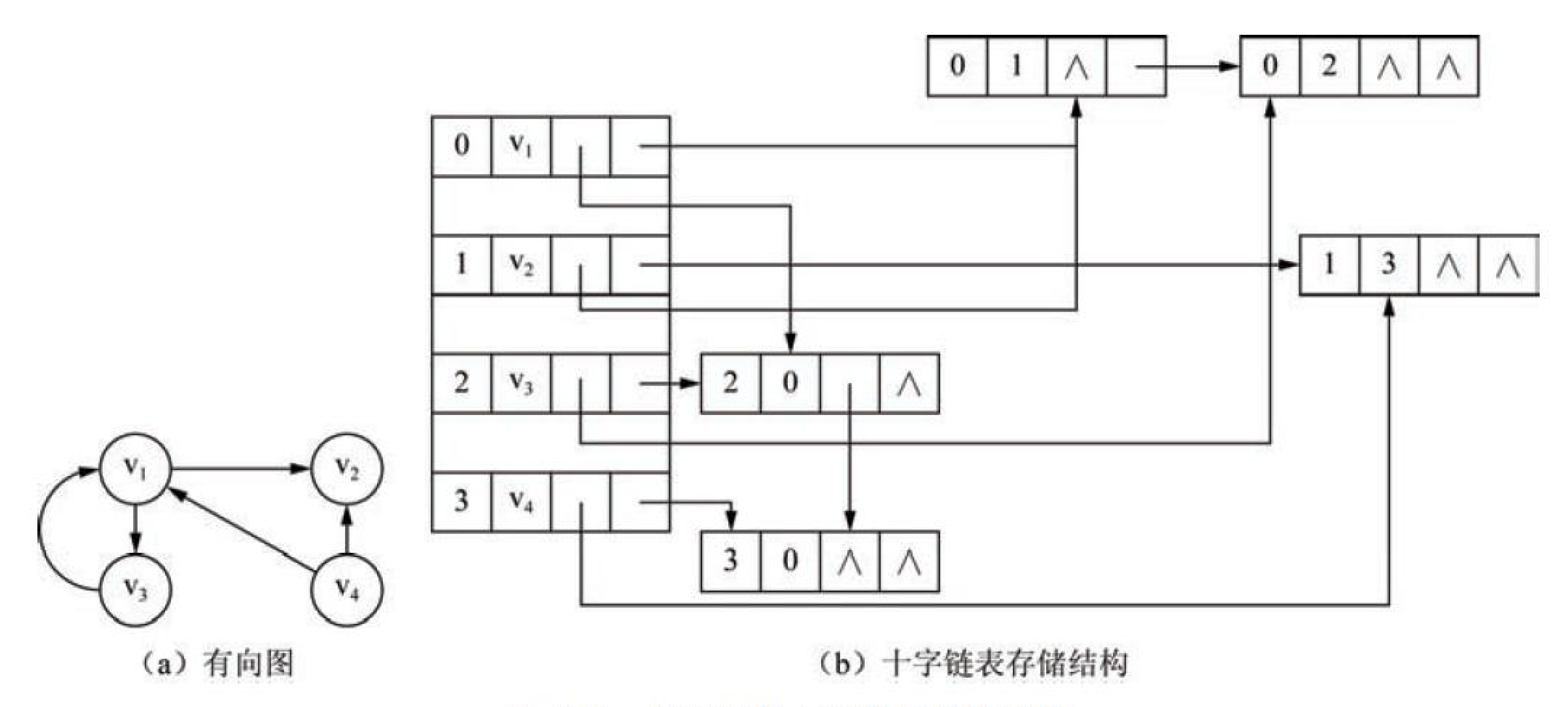


图 4.10　有向图的十字链表存储结构

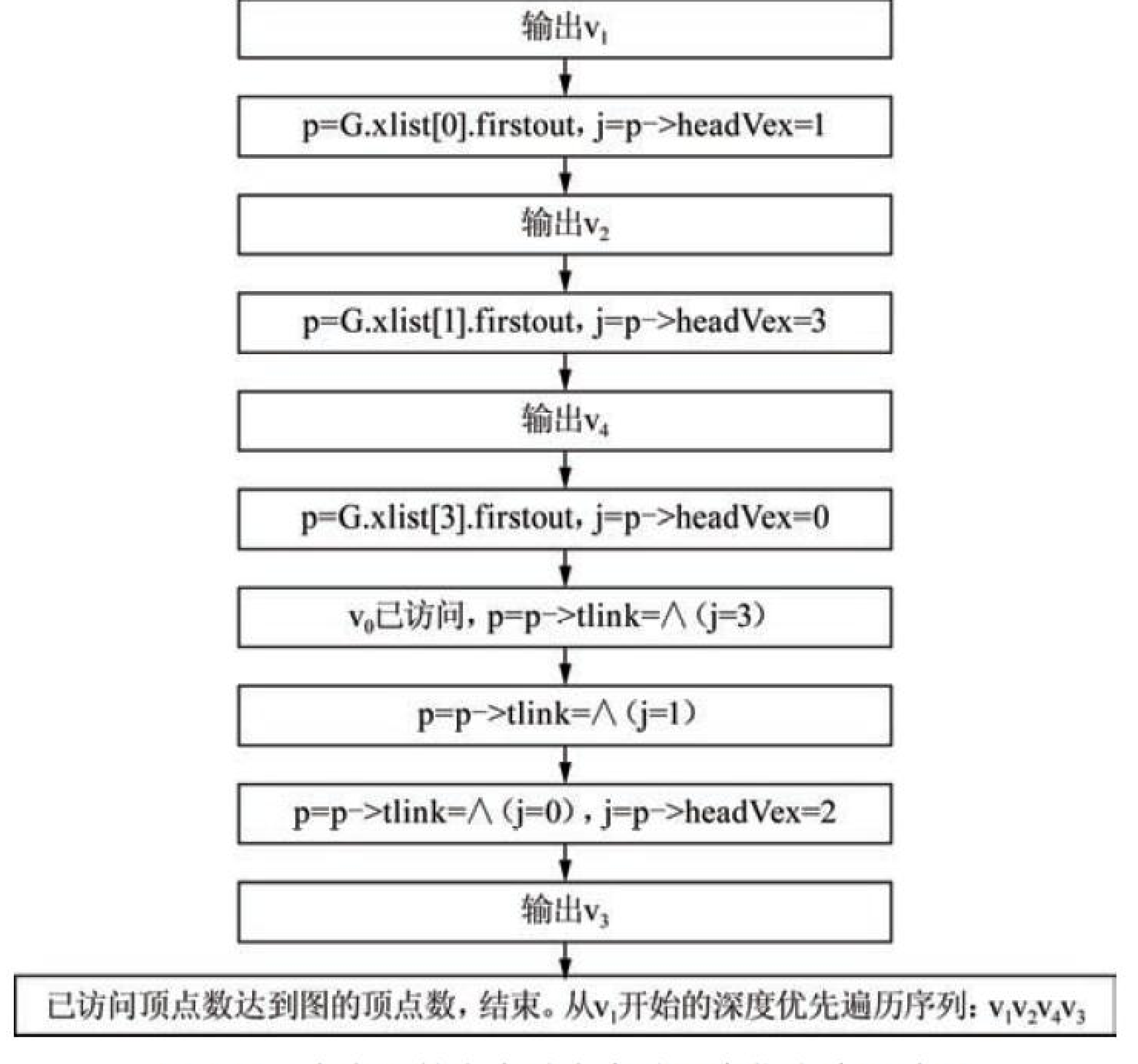


图 4.11　有向图的十字链表存储深度优先遍历过程

3）性能分析

- 深度度优先算法需要辅助堆栈完成遍历，空间复杂度为 O(|V|)。
- 邻接矩阵深度优先遍历时间复杂度为 $O(|V|^2)$，链式存储结构时间复杂度为 O(|V|+|E|)。

4.4.2　广度优先搜索

广度优先遍历类似于树的按层次遍历，是树的层次遍历的推广。

假设图中所有顶点均未被访问，则广度优先搜索过程如下。

（1）从图中某顶点 v 出发，访问此顶点。

（2）依次访问 v 的各个未曾访问过的邻接点。

（3）从这些邻接点出发依次访问它们的邻接点，并使得“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问（通过队列实现），直到图中所有已被访问的顶点邻接点都被访问到。

（4）若此时图中尚有顶点未被访问，则选择图中一个未曾被访问的顶点作起始点，重复上述过程，直到所有顶点都被访问到为止。

注意（同深度优先遍历）：

- 如果仅给出图的逻辑结构而没给具体存储结构，则其广度优先遍历序列不唯一。
- 存储结构一旦确定并给出，遍历序列唯一。

广度优先遍历算法的描述具体如下。

1. 邻接矩阵存储结构

1）算法描述（增加参数：开始顶点，深度优先的指定开始顶点算法自行仿照编写）

```
Boolean visited[MAX_VEX_NUM];          // 是否访问标志数组
#define  TRUE   1
#define  FALSE  0
void BFSTraverse(matrixGraph  G, vertexType  v) {
    for (i=0; i<G.vexNum; i++)
        visited[i]=FALSE;              // 访问标志数组初始化
    InitQueue(Q);                      // 初始化空队 Q
    i= locateVex(G, v);                // 获取顶点 v 在图 G 中的位置
    EnterQueue(Q,i);                   // v 入队
    while (!Empty(Q)) {
        DeleteQueue(Q,i);              // 队首出队
        visit(i);
        visited[i]=TRUE;               // 访问第 i 个顶点 v 并标记
        for(j=0;j<vexNum;j++)          // 将 v 的未访问邻接点编号一一入队
            if (!visited[j] && (G.arcs[i][j]!=0 || G.arcs[i][j]!=INFINITY))
                EnterQueue(Q,j)        // j 未访问入队
    }
}
```

2）示例

某有向图如图 4.12（a）所示，图 4.12（b）为其邻接矩阵，从 v_1 开始的宽度优先遍历过程如图 4.13 所示。

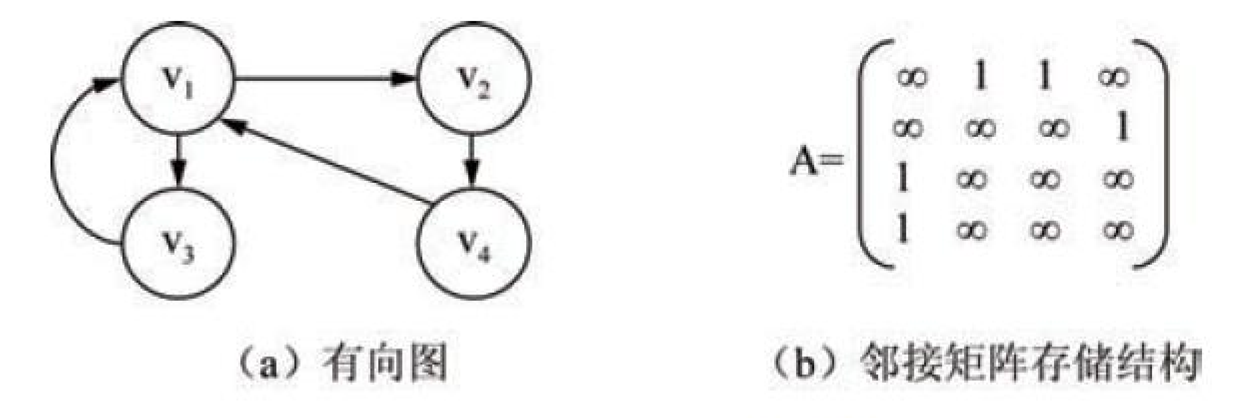


图 4.12　有向图的邻接矩阵存储

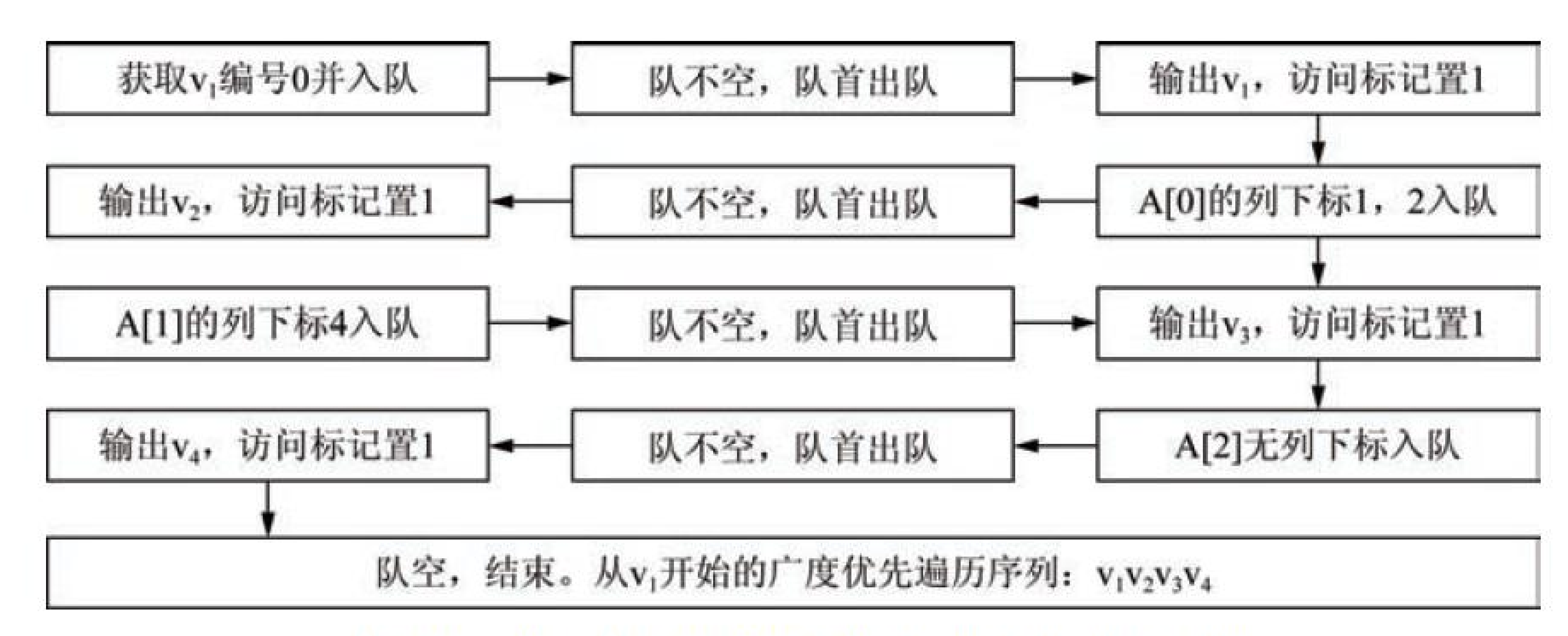


图 4.13　有向图的邻接矩阵存储广度优先遍历过程

2. 邻接表存储结构

1）算法描述

```
Boolean visited[MAX_VEX_NUM];          // 是否访问标志数组
#define  TRUE   1
#define  FALSE  0
void BFSTraverse(adjListGraph  G, vertexType  v) {
    InitQueue(Q);                      // 初始化空队 Q
    i= locateVex(G, v);                // 获取顶点 v 在图 G 中的位置
    EnterQueue(Q,i);                   // v 的位置信息入队
    for (i=0; i<G.vexNum; i++)
        visited[i]=FALSE;              // 访问标志数组初始化
    while (!Empty(Q)) {
        DeleteQueue(Q,i);              // 队首出队
        visit(i);
        visited[i]=TRUE;               // 访问第 i 个顶点并标记
        p =G.vertex[i].firstarc;       // 取第 i 个顶点的邻接顶点
        while(p!=NULL) {
            j=p->adjvex;               // 取第 i 个顶点的邻接点编号 j
             if(!visited[j])
                EnterQueue(Q,j)        // j 未访问入队
                p=p->nextarc;          // 取下一个邻接边结点
        }
    }
}
```

2）示例

某有向图如图 4.14（a）所示，图 4.14（b）为其邻接矩阵，从 v_1 开始的广度优先遍历过程如图 4.15 所示。

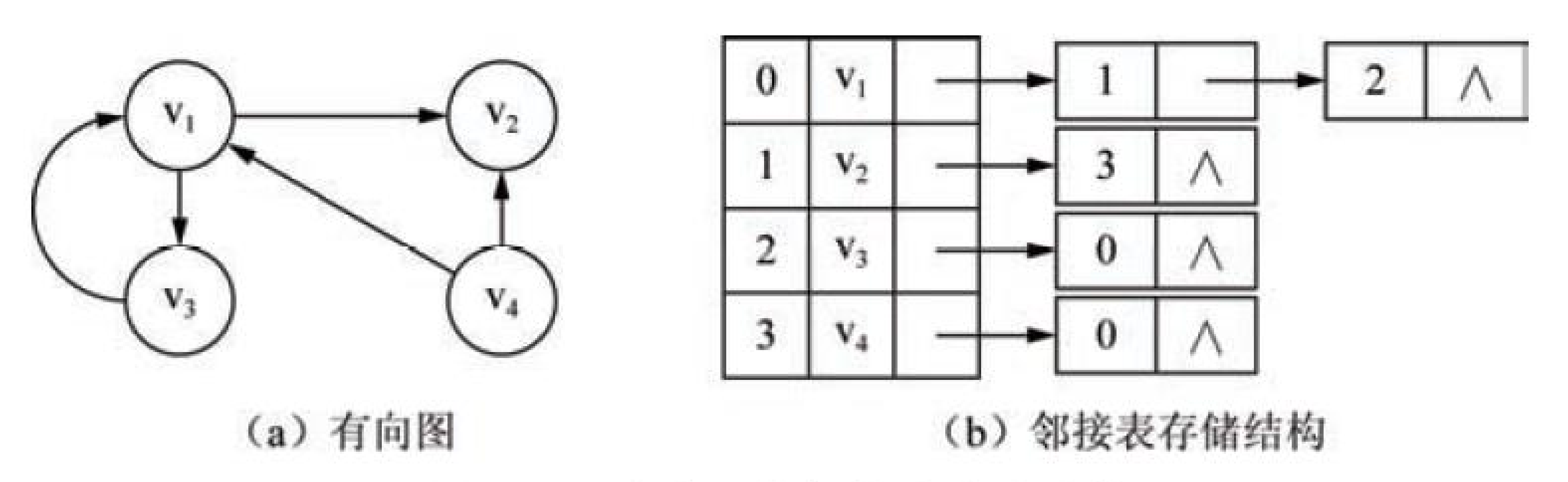


图 4.14　有向图的邻接表存储结构

3. 十字链表存储结构

1）算法描述

```
Boolean visited[MAX_VEX_NUM];          // 是否访问标志数组
#define TRUE   1
#define FALSE 0
void BFSTraverse(crossLinkGraph  G, vertexType  v) {
    InitQueue(Q);                      // 初始化空队 Q
    i= locateVex(G, v);                // 获取顶点 v 在图 G 中的位置
    EnterQueue(Q,i);                   // v 的位置信息入队
    for (i=0; i<G.vexNum; i++)
        visited[i]=FALSE;              // 访问标志数组初始化
    while (!Empty(Q)) {
        DeleteQueue(Q,i);              // 队首出队
        visit(i);
        visited[i]=TRUE;               // 访问第 i 个顶点并标记
        p= G.xlist[i].firstout;        // 取第 i 个顶点的第一个邻接顶点
        while(p!=NULL) {
            j=p->headvex;              // 取第 i 个顶点的邻接点编号 j
            if(!visited[j])
                EnterQueue(Q,j)        // j 未访问入队
                p=p->tlink;            // 取下一个邻接边结点
        }
    }
}
```

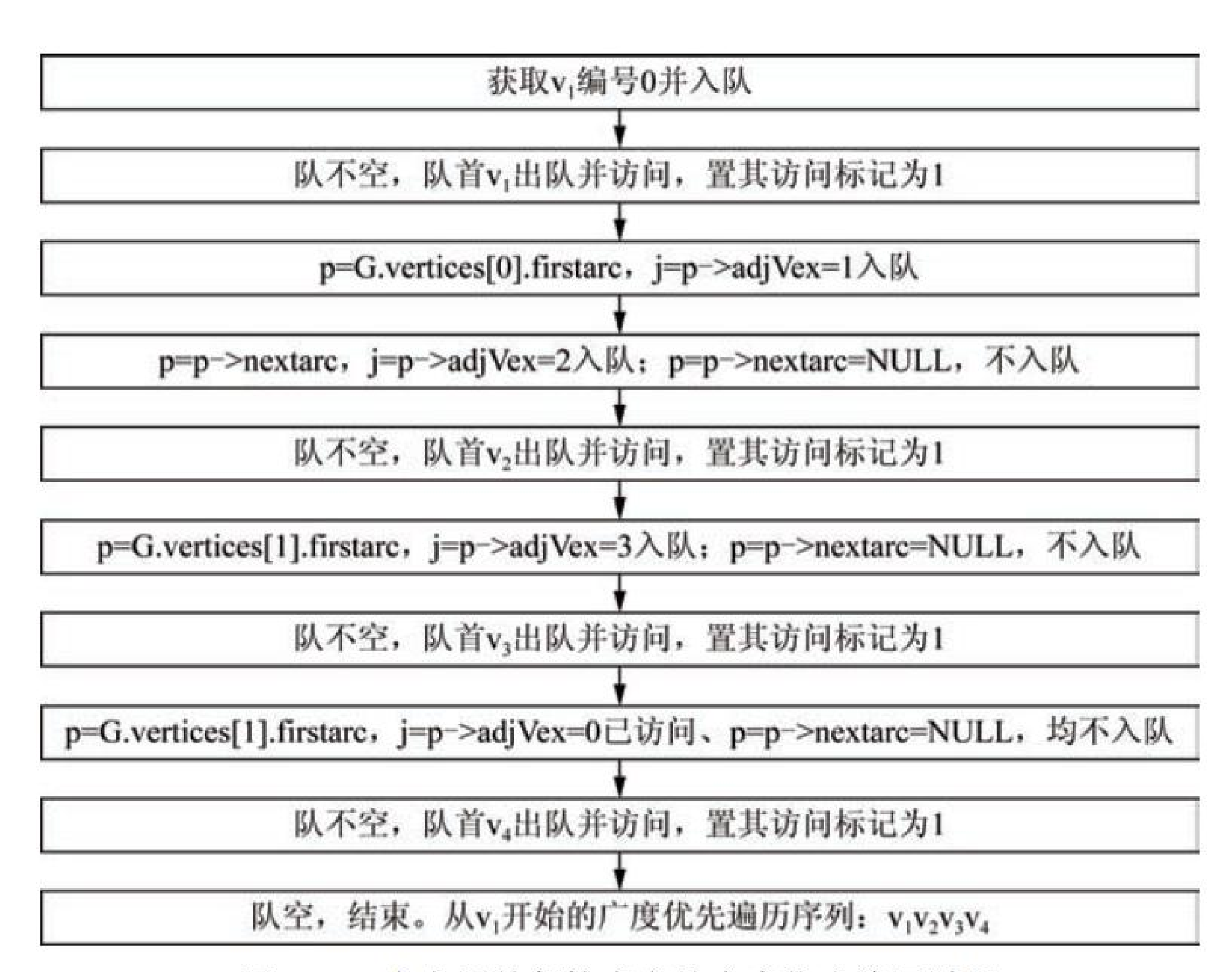


图 4.15　有向图的邻接表存储广度优先遍历过程

2）示例

某有向图如图 4.16（a）所示，图 4.16（b）为其邻接矩阵，从 v_1 开始的深度优先遍历过程如图 4.17 所示。

3）性能分析

① 广度优先算法需要辅助队列完成遍历，所有顶点均需入队，和存储结构无关，空间复杂度为 O(|V|)。

② 邻接矩阵广度优先遍历时间复杂度为 $O(|V|^2)$，链式存储结构时间复杂度为 O(|V|+|E|)，分别适合稠密图和稀疏图。

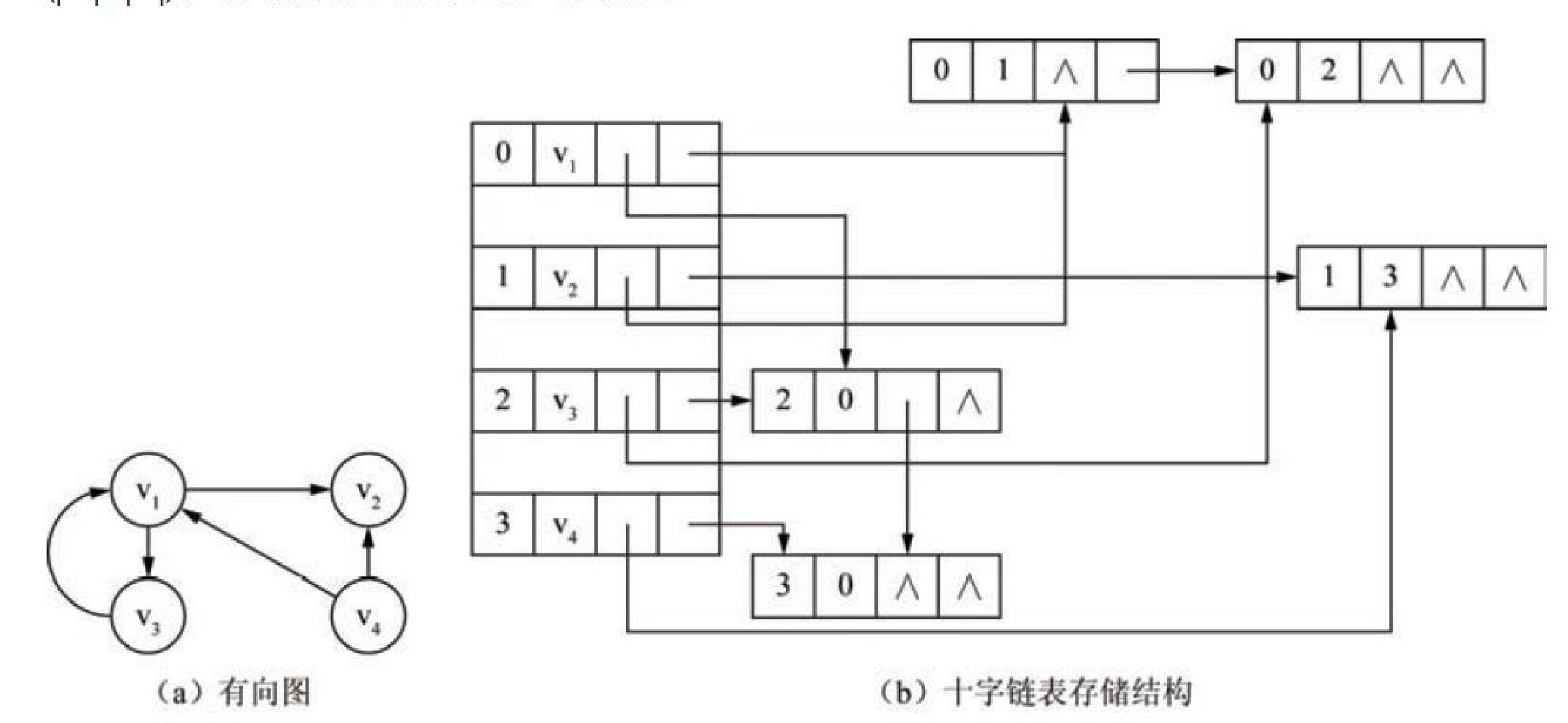


图 4.16　有向图的十字链表存储结构

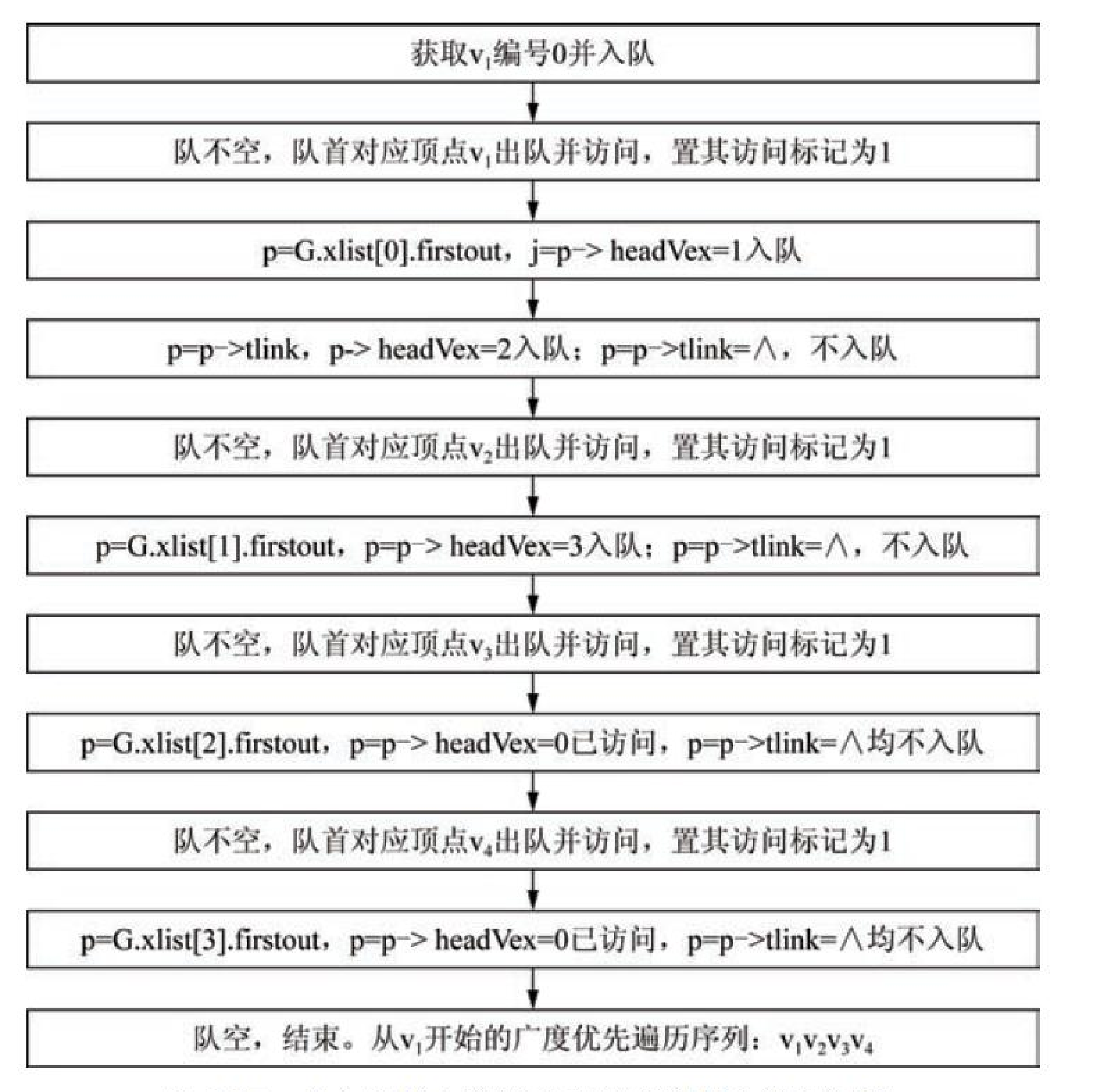


图 4.17　有向图的十字链表存储广度优先遍历过程

4.5 最小生成树

连通图/网（带权图一般称网）的生成树指包含网的所有顶点、n−1 条满足树特征的边所形成的树。n 个顶点的连通网可以构建许多不同的生成树。最小生成树（MST）指所有生成树中权值之和最小的树。构建连通网的最小生成树是图论的基本问题之一，在现实生活中有大量的应用，比如建设交通网络，设计多城市间最短交通路径问题、多地旅游的最佳路线问题等。

最小生成树的特点有以下两点。

（1）最小生成树的权值和为所有生成树的最小值，故权值和唯一。

（2）最小生成树不一定唯一。

常见求解最小生成树的方法有两种。普里姆（Prim）算法和克鲁斯卡尔（Kruskal）算法。

4.5.1 普里姆算法

1．算法概述

普里姆算法构建最小生成树的过程即为依次增大最小生成树的过程，具体如下。

（1）任意选择一个顶点，并将其加入集合 U，则该顶点自成连通图。

（2）在集合 U 和集合 V−U 中选择两个集合顶点之间相连的所有边的最小边，将这条边加入最小生成树，同时将 V−U 中和该边相关联的顶点去除，并将其加入 U。

（3）重复（2），直到 U 中包含连通网的所有顶点。

图 4.18 为普里姆算法构造最小生成树的过程描述，假设从 v_1 开始构建最小生成树，本算法基于图 4.18（a）。

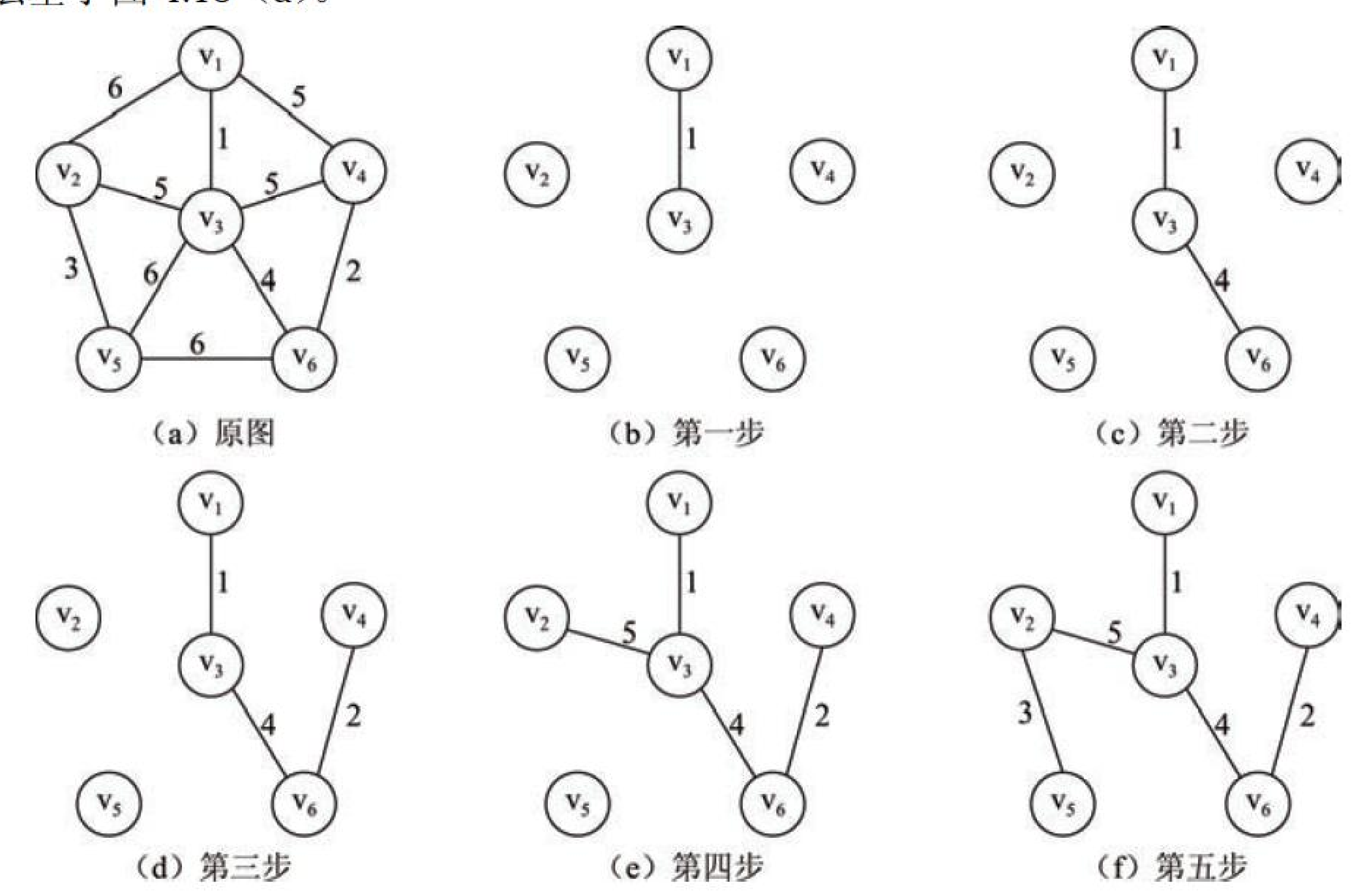


图 4.18 普里姆算法构造最小生成树

图 4.19 为集合 U 和 V−U 的变化过程。

步骤	U	V−U
(0)	$\{v_1\}$	$\{v_2, v_3, v_4, v_5, v_6\}$
(1)	$\{v_1, v_3\}$	$\{v_2, v_4, v_5, v_6\}$
(2)	$\{v_1, v_3, v_6\}$	$\{v_2, v_4, v_5\}$
(3)	$\{v_1, v_3, v_6, v_4\}$	$\{v_2, v_5\}$
(4)	$\{v_1, v_3, v_6, v_4, v_2\}$	$\{v_5\}$
(5)	$\{v_1, v_3, v_6, v_4, v_2, v_5\}$	{ }

图 4.19　普里姆算法构造最小生成树集合 U 和 V−U 的变化过程

2. 算法实现

1）算法流程

（1）假设从 v_1 开始构建最小生成树，则 U=$\{v_1\}$，V−U=$\{v_2, v_3, v_4, v_5, v_6\}$。

（2）初始化数组 closedge[|V|]，存储 U 中各顶点到 V−U 中各顶点之间的最小权值的边的权值。

（3）在 closedge[]中找到一条权值最短的边，假设与此边相关联的 V−U 中的另一个顶点序号为 k，则将该边加入最小生成树，同时 v_k 从集合 V−U 移入 U，并令 closedge[k].lowcost=0，表明 k 已加入 U。

（4）修改 V−U 中所有顶点的 closedge 值为 U 中各顶点到 V−U 中各顶点之间的最小权值的边的权值。

（5）重复（3）和（4），直到 U=V。

2）伪代码

以邻接矩阵存储结构为例，代码如下。

```
struct {
    int  adjvex;
    int  used;
    int  lowcost;
}closedge[MAX_VEX_NUM];
void primMiniTree(matrixGraph  G, vertexType  v) {
                        // 从顶点 v 开始利用普里姆算法构建 G 的最小生成树
    i=locateVex(G, v);
    for(j=0;j< G.vexNum; j++ ) {
        closedge[j].lowcost = G.arcs[i][j].adj;    // v 到其他顶点的初始权值
                                                   // 为邻接矩阵第 i 行的值
        closedge[j].adjvex=i;           // adjvex 保存 U 中到 V-U 最小边的顶点编号
        closedge[j].used=0;             // used=0 表示第 j 个顶点在 V-U 中
    }
    closedge[i].used=1;                 // 顶点 v 放入 U
    for( i=0; i<G.vexNum-1; i++ ) {
        j=0;
        min=INFINITY;
        for( k=1; k<G.vexNum; k++ )
            if( (closedge[k].used ==0) && (closedge[k].lowcost<min) ) {
```

```
                min=closedge[k].lowcost;
                j=k;
            }
        closedge[j].used=1;              // 第 j 个顶点加入 U
        for( k=0; k< G.vexNum; k++ )
                //由于第 j 个顶点加入 U，更新 V-U 中各顶点 lowcost 和 adjvex 成员的值
            if(      (closedge[k].used==0)       &&       (G.arcs[k][j].adj
        <closedge[k].lowcost) ) {
                closedge[k].lowcost=G.arcs[k][j].adj;
                closedge[k].adjvex=j;
            }
    }
}
```

3）算法执行演示

算法执行的过程信息如图 4.20 所示。

closedge \ 顶点i	2	3	4	5	6	U	V-U
adjvex lowcost	v_1 6	v_1 1	v_1 5			$\{v_1\}$	$\{v_2, v_3, v_4, v_5, v_6\}$
adjvex lowcost	v_3 5	0	v_1 5	v_3 6	v_3 4	$\{v_1, v_3\}$	$\{v_2, v_4, v_5, v_6\}$
adjvex lowcost	v_3 5	0	v_6 2	v_3 6	0	$\{v_1, v_3, v_6\}$	$\{v_2, v_4, v_5\}$
adjvex lowcost	v_3 5	0	0	v_3 6	0	$\{v_1, v_3, v_6, v_4\}$	$\{v_2, v_5\}$
adjvex lowcost	0	0	0	v_2 3	0	$\{v_1, v_3, v_6, v_4, v_2\}$	$\{v_5\}$
adjvex lowcost	0	0	0	0	0	$\{v_1, v_3, v_6, v_4, v_2, v_5\}$	$\{\}$

图 4.20　普里姆算法演示

4）算法分析

普里姆算法为顶点之间的最短关系查找，时间复杂度为 $O(|V|^2)$，适合稠密图求最小生成树。

4.5.2　克鲁斯卡尔算法

1．算法概述

克鲁斯卡尔（Kruskal）算法构建最小生成树的过程即依次挑选最小权值边的过程，具体如下。

（1）将原图中的顶点复制得到一个新的无边网，即包含 |E| 个连通分量的非连通图。

（2）原图中选择一条满足如下两个条件的边：

① 权值最小。

② 该边连接新网的两个连通分量，构成更大的连通无环网。

（3）原图中删除该边。

（4）重复（2）和（3），直到新网为包含所有顶点的一个连通分量，即新网含 |E| 个顶点、|E|−1 条边。

图 4.21 为克鲁斯卡尔算法构造最小生成树的过程描述，本算法基于图 4.21（a）。

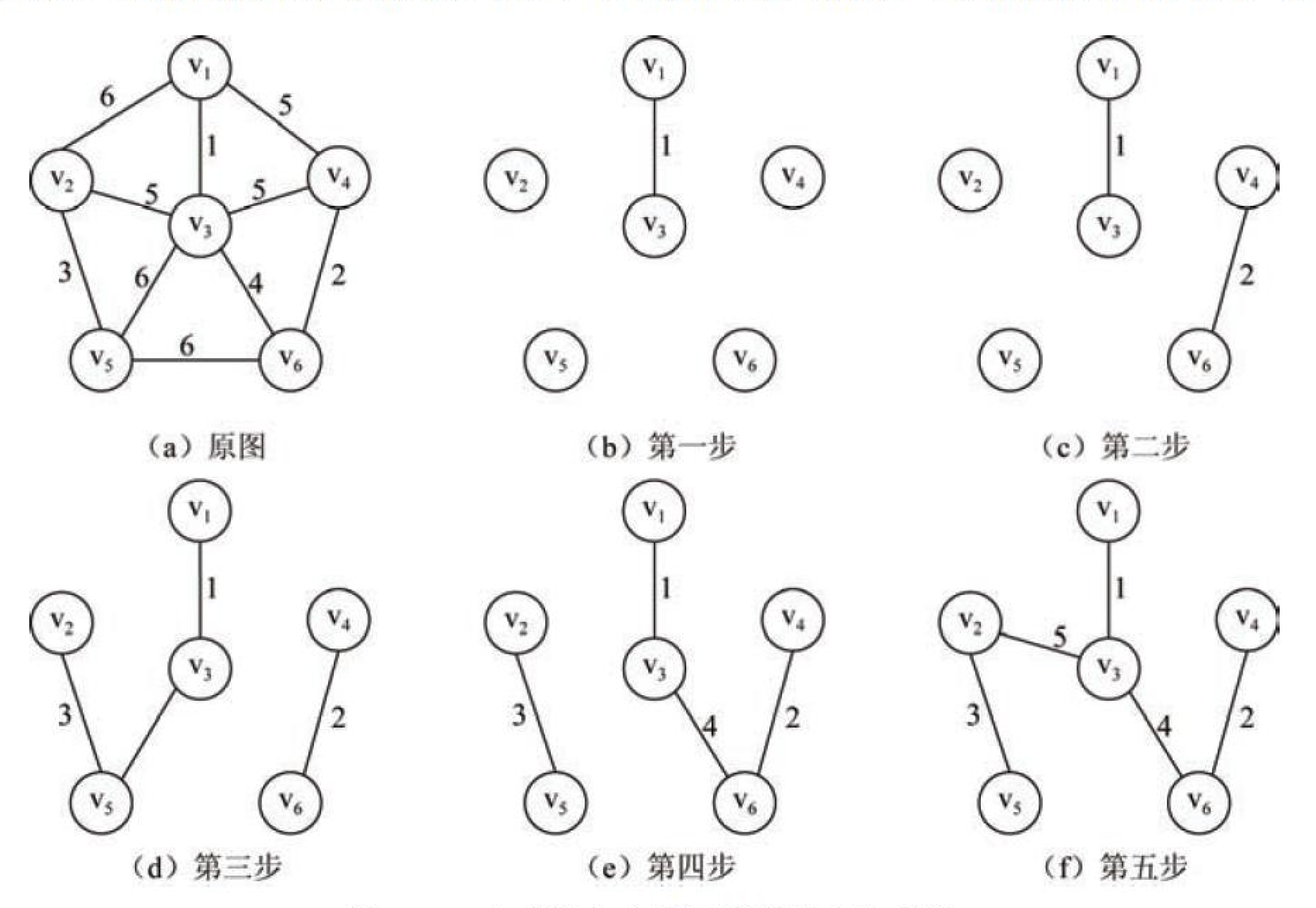


图 4.21　克鲁斯卡尔算法构造最小生成树

2．算法实现

1）算法流程

（1）将原图（网）中所有边的权值升序排列。如图 4.21（a）中的升序结果为 qa={1，2，3，4，5，5，5，6，6，6}。

（2）原图所有顶点构成新的无边网。

（3）图 4.21（a）中最小边 1 加入无边网，如图 4.21（b）所示；从图 4.21（a）中删除该边，则 qa={2，3，4，5，5，5，6，6，6}。

（4）图 4.21（a）中最小边 1、2 加入无边网，如图 4.21（c）所示；从图 4.21（a）中删除该边，则 qa={3，4，5，5，5，6，6，6}。

（5）图 4.21（a）中最小边 3 加入无边网，如图 4.21（d）所示；从图 4.21（a）中删除该边，则 qa={4，5，5，5，6，6，6}。

（6）图 4.21（a）中最小边 4 加入无边网，如图 4.21（e）所示；从图 4.21（a）中删除该边，则 qa={5，5，5，6，6，6}。

（7）图 4.21（a）中共有 3 条权值为 5 的边，其中边 (v_1,v_4) 和 (v_3,v_4) 将使无边网形成回路，边 (v_2,v_3) 不构成回路，加入无边网，如图 4.21（f）所示，此时最小生成树需要的 |E|−1 条边选择完毕，算法结束。

2）伪代码

以邻接矩阵存储结构为例的克鲁斯卡尔算法的代码如下。

```
#define  MAX_ARC_NUM  (MAX_VEX_NUM*(MAX_VEX_NUM-1))
                                    // 定义最多边数 MAX_ARC_NUM
typedef struct KMT{
    int  startVex;
    int  endVex;
    int  weight;
} kruskalMiniTree[MAX_ VEX _NUM - 1];
void kruskalMiniTree(matrixGraph G) {
    struct  KMT  KMTarcs[MAX_ARC_NUM];
    k=0;
    for(i=0;i< G.vexNum;i++ )     // 将原图所有边的信息保存于 KMTarcs 数组
        for(j=0;j< G.vexNum;j++ )
            if(G.arcs[i][j]!=0 || G.arcs[i][j]!=INFINITY){
                KMTarcs[k].startVex=i;
                KMTarcs[k].endVex=j;
                KMTarcs[k].weight=G.arcs[i][j];
                k++;
            }
    for(i=0; i<k-1; i++ ) {       // KMTarcs 数组元素按 weight 成员的值升序排列
        change=0;
        for(j=0; j<k-i-1; j++ )
            if(KMTarcs[j].weight > KMTarcs[j+1].weight) {
                t1=KMTarcs[j].startVex;
                t2=KMTarcs[j].endVex;
                t3= KMTarcs[j].weight
                KMTarcs[j].startVex=KMTarcs[j+1].startVex;
                KMTarcs[j].endVex=KMTarcs[j+1].endVex;
                KMTarcs[j].weight=KMTarcs[j+1].weight
                KMTarcs[j+1].startVex=t1;
                KMTarcs[j+1].endVex=t2;
                KMTarcs[j+1].weight=t3;
                change=1;
            }
        if( !change )
            break;
}
i=0;
for( j=0; j<k; j++ ){
                        // 挑选不构成回路的 G.vexNum-1 条最小边构造最小生成树
        if( kruskalMiniTree[0-j] 对应网不构成回路 ){
            kruskalMiniTree[i].startVex =KMTarcs[j].startVex;
            kruskalMiniTree[i].endVex=KMTarcs[j].endVex;
            kruskalMiniTree[i].weight= KMTarcs[j].weight;
            i++;
        }
        if( i==G.vexNum-1)
            break;
    }
}
```

3）算法分析

克鲁斯卡尔算法为最小边的查找过程，构建最小生成树的时间复杂度为 $O(|E|*\log_2^{|E|})$，适合稀疏图求最小生成树。

4.6 最短路径

带权图中任意两个顶点 A 和 B 之间可能存在多条路径，权值之和（路径长度）最小的称为最短路径。求解图的最短路径是图论的基本问题之一，在现实生活中有大量的应用，比如两城市之间距离最短或花费最少的最佳路线等。最短路径有以下两个特点。

（1）最短短路径上的权值之和为所有路径中权值之和的最小值，故权值和唯一。

（2）最短路径不一定唯一。

有两种常见的最短路径问题，一是单源顶点到图中其余各顶点的最短路径；二是图中任意两顶点之间的最短路径。

4.6.1　单源最短路径

单源最短路径是已知有向图（网）G=（V,E）和一个源顶点 v_i，分别求 v_i 到顶点 v_j 之间权值之和最小的路径及其长度。常用的求解方法为 Dijkstra 算法。

1．Dijkstra 算法概述

算法本质：路径长度依次递增，逐步生成最短路径。该算法基于最短路径的以下性质。

（1）假设 $<v_0, v_1>$ 为权值最小的弧，则长度最短的路径必含 $<v_0, v_1>$。

（2）长度次短的路径满足以下两个条件的任意一个：

① 从源点通过一条弧 $<v_0, v_2>$ 到达 v_2。

② 从源点经过两条弧到达 v_2，则这两条弧必为 $<v_0, v_1>$ 和 $<v_1, v_2>$，即经过 v_1。

（3）长度第三短的路径满足以下四个条件中的任意一个。

① 从源点通过一条弧 $<v_0, v_3>$ 到达 v_3。

② 从源点经过两条弧到达 v_3，则这两条弧为 $<v_0, v_1>$ 和 $<v_1, v_3>$，即经过 v_1。

③ 从源点经过两条弧到达 v_3，则这两条弧为 $<v_0, v_2>$ 和 $<v_2, v_3>$，即经过 v_2。

④ 从源点经过三条弧到达 v_3，则这三条弧为 $<v_0, v_1>$ 、$<v_1, v_2>$和 $<v_2, v_3>$，即经过 v_1 和 v_2。

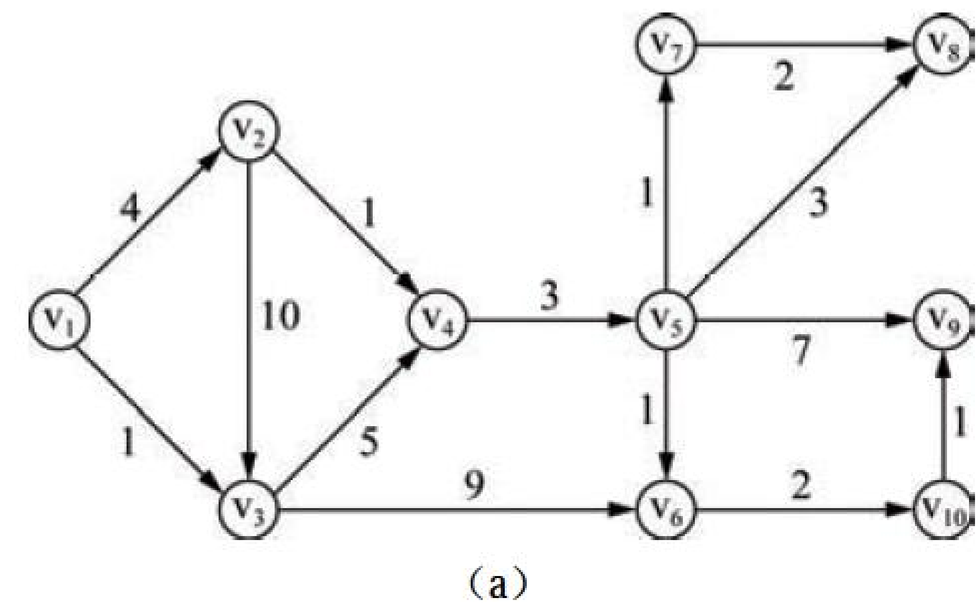
（4）以此类推，求解下一条长度次短的路径，只有以下两类情况：

① 只有一条弧。

② 有多条弧，且最后一条弧的一个端点必为已求得最短路径的顶点，其余弧的两个顶点均为已求得最短路径的顶点。

2．Dijkstra 算法流程

如图 4.22 所示，假设求解起始顶点 v 到其他顶点 v_i 的最短路径。定义数组 D，D[i]表示当前从 v 到 v_i 的最短路径值。求解最短路径的流程即修正 D[i]的过程。



（a）

终点	从 v_1 到各顶点的 D[i]值和最短路径求解过程				
	初始	1	2	3	4
v_2	4 (v_1, v_2)	4			
v_3	1 (v_1, v_3)				
v_4	∞	6 (v_1, v_3, v_4)	5 (v_1, v_2, v_4)		
v_5	∞	∞	∞	8 (v_1, v_2, v_4, v_5)	
v_6	∞	10 (v_1, v_3, v_6)	10 (v_1, v_3, v_6)	10 (v_1, v_3, v_6)	9 $(v_1, v_2, v_4, v_5, v_6)$
v_7	∞	∞	∞	∞	9 $(v_1, v_2, v_4, v_5, v_7)$
v_8	∞	∞	∞	∞	11 $(v_1, v_2, v_4, v_5, v_8)$
v_9	∞	∞	∞	∞	15 $(v_1, v_2, v_4, v_5, v_9)$
v_{10}	∞	∞	∞	∞	∞
本次所求最短路径顶点	v_3	v_2	v_4	v_5	v_6
求得的最短路径	$<v_1, v_3>$	$<v_1, v_2>$	$<v_1, v_2>$ $<v_2, v_4>$	$<v_1, v_2>$ $<v_2, v_4>$ $<v_4, v_5>$	$<v_1, v_2>$,$<v_2, v_4>$ $<v_4, v_5>$,$<v_5, v_6>$

终点	5	6	7	8	
v_7	9 $(v_1, v_2, v_4, v_5, v_7)$				
v_8	11 $(v_1, v_2, v_4, v_5, v_8)$	11 $(v_1, v_2, v_4, v_5, v_8)$			
v_9	15 $(v_1, v_2, v_4, v_5, v_9)$	15 $(v_1, v_2, v_4, v_5, v_9)$	15 $(v_1, v_2, v_4, v_5, v_9)$	12 $(v_1, v_2, v_4,$ $v_5, v_6, v_{10}, v_9)$	
v_{10}	11 $(v_1, v_2, v_4,$ $v_5, v_6, v_{10})$	11 $(v_1, v_2, v_4,$ $v_5, v_6, v_{10})$	11 $(v_1, v_2, v_4,$ $v_5, v_6, v_{10})$		
本次所求最短路径顶点	v_7	v_8	v_{10}	v_9	
求得的最短路径	$<v_1, v_2>$ $<v_2, v_4>$ $<v_4, v_5>$ $<v_5, v_7>$	$<v_1, v_2>$ $<v_2, v_4>$ $<v_4, v_5>$ $<v_5, v_8>$	$<v_1, v_2>$ $<v_2, v_4>$ $<v_4, v_5>$ $<v_5, v_6>$ $<v_6, v_{10}>$	$<v_1, v_2>$ $<v_2, v_4>$ $<v_4, v_5>$ $<v_5, v_6>$ $<v_6, v_{10}>$ $<v_{10}, v_9>$	

（b）

图 4.22　带权有向图及单源最短路径求解流程

（1）D[i]初值：如果从 v 到 v_i 有弧，则 D[i]为弧上的权值；否则 D[i]的值为∞。

（2）选择值最小的数组元素 D[min]，图中最短的路径即为 v 到 v_{min} 的路径，v_{min} 为已求的从顶点 v 开始的最短路径顶点，路径为< v, v_{min}>。

（3）修改 D[i] (i!=min)：如果 D[i]+ | < v_i, v_j> |< D[j]，则 D[j]= D[i]+ | < v_i, v_j> |，其中 | < v_i, v_j> |表示顶点 v_i 和 v_j 之间的弧长，最短路径由 <…, v_j> 改为 <…, v_i, v_j>。

（4）重复（2）和（3），直到求得 v 到其他所有顶点 v_i 的最短路径。

图 4.22（b）为图 4.22（a）从顶点 v_1 到其余各顶点的最短路径的求解过程。

3．Dijkstra 算法伪代码（以邻接矩阵为例）

```
    void  shortestPathDijkstra(matrixGraph  G,vertexType  v,pathMatrix *pm,
shortestPath *sp){
        i=locateVex(G, v);                  // 获取顶点 v 在无向图 G 中的位置
        for(j=0 ; j<G.vexNum ; j++) {       // 对每个顶点的三个标记数组 final、sp、pm
                                            // 进行初始化
            finished[j]=FALSE ;             // finished [j]表示是否已求得 v 到第 j
                                            // 个顶点最短路径，初始化为假
            sp[j]=G.arcs[i][j].adj ;        // sp[j]表示 v 到第 j 个顶点的最短路径值，初
                                            // 值为 v 到该顶点弧或边的长度
            for(k=0;k<G.vexnum;k++)
                pm[i][k]=FALSE;             // pm[i][k]表示第 i 个顶点是否为 v 到第 k
                                            // 个顶点最短路径上的顶点
            if(sp[j]<INFINITY) {            // 如果 v 到第 j 个顶点有边，则 v 是 v 到第 j
                                            // 个顶点最短路径上的顶点
                pm[i][j]=TRUE;
                pm[j][j]=TRUE;              // 第 j 个顶点是 v 到第 j 个顶点最短路径上的
                                            // 顶点
            }
        }
        sp[i]=0 ;                           // v 到 v 的最短路径长度为 0
        finished[i]=TRUE ;                  // 已求得 v 到 v 的最短路径
        for (j=1; j<G.vexnum ; j++) {       // 求 v 到其他顶点的最短路径
            min= INFINITY;                  // 最小值 min 初始化为系统最大整数值
            for(k=0 ; k<G.vexnum ; k++ )    // 搜索 v 到所有未求得最短路径顶点的路径的
                                            // 最小值
                if ( ! finished[k] )        // 如果第 k 个顶点的最短路径还没求得
                    if ( sp[k]<min ) {      // 如果 v 到第 k 个顶点的当前最短路径长度小
                                            // 于当前最小值
                        h=k ;               // k 的值暂存于 h
                        min=sp[k] ;         // v 到第 k 个顶点的当前最短路径长度存于 min
                    }
                finished[h]=TRUE;
                                            // v 到第 h 个顶点的当前路径长度最短，即 v
                                            // 到第 h 个顶点的最短路径已找到
                for (t=0 ; t<G.vexnum ; t++ )
                                            // 由于 v 到第 h 个顶点的最短路径已找到，修
                                            // 正其他未求得最短路径顶点的最短路径
                    if ( ! finished[t]&& ( sp[t]>min+G.arcs[h][t] ) ) {
                                            // 如果第 t 个顶点的最短路径 sp[t]还没求得，
                                            //并且 sp[t]大于 v 到第 h 个顶点的最短路径
```

```
                                   // 长度+第 h 个顶点到第 t 个顶点的弧或边长
                    sp[t]= min+G.arcs[h][t] ;
                                   // 修改 sp[t]为 v 到第 h 个顶点的最短路径长度+
                                   // 第 h 个顶点到第 t 个顶点的弧或边长
                    pm[t]=pm[h];   // 第 h 个顶点的最短路径上的顶点赋给第 t 个顶
                                   // 点的最短路径
                pm[t][t]=TRUE;     // 第 t 个顶点是 v 到第 t 个顶点最短路径上的顶点
                                   // 即第 t 个顶点加入 v 到第 t 个顶点最短路径
            }
        }
    }
```

4．Dijkstra 算法分析

以邻接矩阵为存储结构，Dijkstra 算法的时间复杂度为 $O(|V|^2)$，对于稀疏图，应选择邻接表作为存储结构，Dijkstra 算法的时间复杂度为 $O(|E|)$。

Dijkstra 算法假设所有弧或边的权值非负，不考虑权值为负的情况。

4.6.2　任意两个顶点之间的最短路径

任意两个顶点之间的最短路径是已知带权有向图（网）G=（V,E），分别求任一对顶点 v_i 到 v_j 之间权值和最小的路径及其长度（$i\neq j$）。常见求解方法为 Floyd 算法。

也可以使用 Dijkstra 算法求解本节问题，再加一层外循环，使所有顶点依次为源点，重复执行上节算法即可，时间复杂度为 $O(|V|^3)$。

1．Floyd 算法流程

设有带权有向图 G 的存储结构为邻接矩阵，求解 G 中任意一对顶点 v_i 到 v_j 的最短路径流程如下（最短路径的长度值用 shortestPath(i, j,k) 表示，shortestPath(i, j,k)为从 v_i 到 v_j 的、中间顶点个数不多于 k 个的最短路径长度值）。

（1）如果从 v_i 到 v_j 有弧，则从 v_i 到 v_j 存在一条长度为 G.arcs[i][j]的直接路径（v_i, v_j）（未必是最短路径，需进行最多|V| −1 次比较才能得到最终结果），该路径不经过任何中间顶点。直接路径（v_i, v_j）记为 path(i, j,0)，且 shortestPath[0][i][j]=G.arcs[i][j].adj。

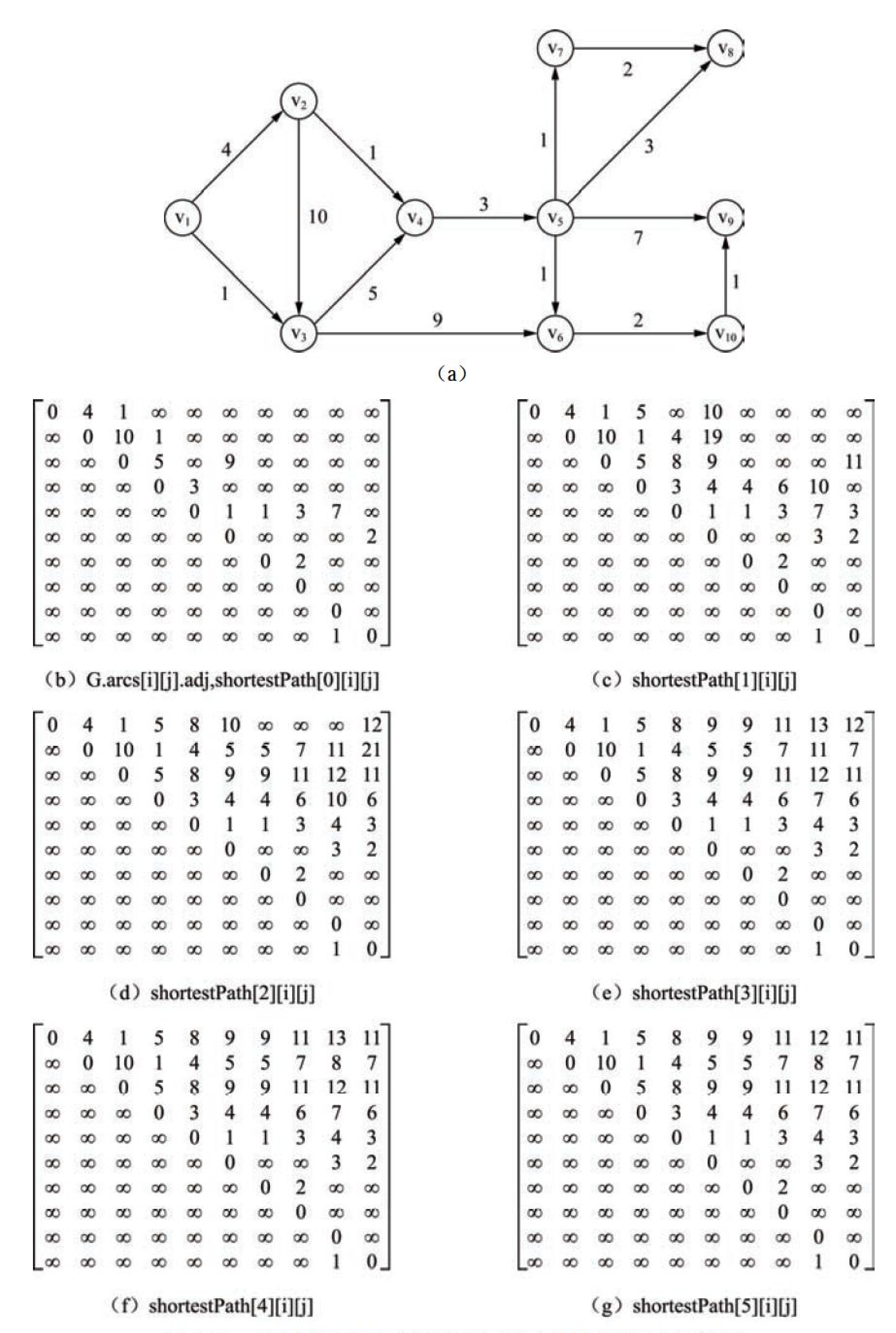
（2）判断路径（v_i, v_t, v_j）是否存在（即判断弧 $<v_i, v_t>$ 及 $<v_t, v_j>$ 是否存在）。如果存在，则将路径（v_i, v_j）和（v_i, v_t, v_j）的长度值较小者作为从 v_i 到 v_j 的、中间顶点个数不多于 1 个的最短路径 shortestPath[1][i][j]，该路径记为 path(i, j,1)。

（3）继续在路径上依次增加顶点，设增加的顶点个数为 k，k∈[2, |V|−2]，如果 path(i, j, k−1)和 path(i, j, k) 均存在，则将 shortestPath[k−1][i][j] 和 min(| shortestPath[k−1][i][t] | + | (v_t,v_j) |) 的较小者作为从 v_i 到 v_j 的、中间顶点个数不多于 k 个的最短路径长度值，记为 shortestPath[k][i][j]。

（4）最多经过 |V| −1 次比较，最后求得的 shortestPath[|V| −2][i][j] 即从 v_i 到 v_j 的最短路径。

注意：设带权有向图 G 中任意一对顶点 v_i 到 v_j 的中间顶点个数最多为 MAX 个，则求解最短路径 MAX+1 次，比较 MAX 次即可找到任意两顶点之间的最短路径，即求解出 shortestPath[MAX][i][j]，算法结束。

图 4.2（b）～图 4.23（h）为图 4.23（a）从顶点 v_1 到其余各顶点的最短路径的求解过程。



（a）

0	4	1	∞	∞	∞	∞	∞	∞	∞
∞	0	10	1	∞	∞	∞	∞	∞	∞
∞	∞	0	5	∞	9	∞	∞	∞	∞
∞	∞	∞	0	3	∞	∞	∞	∞	∞
∞	∞	∞	∞	0	1	1	3	7	∞
∞	∞	∞	∞	∞	0	∞	∞	∞	2
∞	∞	∞	∞	∞	∞	0	2	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	0	∞
∞	∞	∞	∞	∞	∞	∞	∞	1	0

（b）G.arcs[i][j].adj,shortestPath[0][i][j]

0	4	1	5	∞	10	∞	∞	∞	∞
∞	0	10	1	4	19	∞	∞	∞	∞
∞	∞	0	5	8	9	∞	∞	∞	11
∞	∞	∞	0	3	4	4	6	10	∞
∞	∞	∞	∞	0	1	1	3	7	3
∞	∞	∞	∞	∞	0	∞	∞	3	2
∞	∞	∞	∞	∞	∞	0	2	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	0	∞
∞	∞	∞	∞	∞	∞	∞	∞	1	0

（c）shortestPath[1][i][j]

0	4	1	5	8	10	∞	∞	∞	12
∞	0	10	1	4	5	5	7	11	21
∞	∞	0	5	8	9	9	11	12	11
∞	∞	∞	0	3	4	4	6	10	6
∞	∞	∞	∞	0	1	1	3	4	3
∞	∞	∞	∞	∞	0	∞	∞	3	2
∞	∞	∞	∞	∞	∞	0	2	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	0	∞
∞	∞	∞	∞	∞	∞	∞	∞	1	0

（d）shortestPath[2][i][j]

0	4	1	5	8	9	9	11	13	12
∞	0	10	1	4	5	5	7	11	7
∞	∞	0	5	8	9	9	11	12	11
∞	∞	∞	0	3	4	4	6	7	6
∞	∞	∞	∞	0	1	1	3	4	3
∞	∞	∞	∞	∞	0	∞	∞	3	2
∞	∞	∞	∞	∞	∞	0	2	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	0	∞
∞	∞	∞	∞	∞	∞	∞	∞	1	0

（e）shortestPath[3][i][j]

0	4	1	5	8	9	9	11	13	11
∞	0	10	1	4	5	5	7	8	7
∞	∞	0	5	8	9	9	11	12	11
∞	∞	∞	0	3	4	4	6	7	6
∞	∞	∞	∞	0	1	1	3	4	3
∞	∞	∞	∞	∞	0	∞	∞	3	2
∞	∞	∞	∞	∞	∞	0	2	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	0	∞
∞	∞	∞	∞	∞	∞	∞	∞	1	0

（f）shortestPath[4][i][j]

0	4	1	5	8	9	9	11	12	11
∞	0	10	1	4	5	5	7	8	7
∞	∞	0	5	8	9	9	11	12	11
∞	∞	∞	0	3	4	4	6	7	6
∞	∞	∞	∞	0	1	1	3	4	3
∞	∞	∞	∞	∞	0	∞	∞	3	2
∞	∞	∞	∞	∞	∞	0	2	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	0	∞
∞	∞	∞	∞	∞	∞	∞	∞	1	0

（g）shortestPath[5][i][j]

图 4.23　带权有向图及任意两顶点之间最短路径求解流程

$$\begin{bmatrix} 0 & 4 & 1 & 5 & 8 & 9 & 9 & 11 & 12 & 11 \\ \infty & 0 & 10 & 1 & 4 & 5 & 5 & 7 & 8 & 7 \\ \infty & \infty & 0 & 5 & 8 & 9 & 9 & 11 & 12 & 11 \\ \infty & \infty & \infty & 0 & 3 & 4 & 4 & 6 & 7 & 6 \\ \infty & \infty & \infty & \infty & 0 & 1 & 1 & 3 & 4 & 3 \\ \infty & \infty & \infty & \infty & \infty & 0 & \infty & \infty & 3 & 2 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 & 2 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{bmatrix}$$

（h）shortestPath[6][i][j]

图 4.23（续）

由于图 4.23（a）中任意两顶点间包含顶点数最多的路径为从 v_1 到 v_9 的路径（$v_1, v_2, v_3, v_4, v_5, v_6, v_{10}, v_9$），共含 6 个顶点 v_2、v_3、v_4、v_5、v_6、v_{10}，所以 k=6 即可结束算法，shortestPath[6][i][j]的值即为从 v_i 到 v_j 的最短路径。

2．Floyd 算法伪代码描述（以邻接矩阵为例）

```
void shortestPathFloyd (matrixGraph G, int shortestPath[m][n][h]){
   MAX= G 中任意两顶点间包含顶点数最多的路径长度 - 2;
                                          // 可利用图的遍历算法求解
   for(i=0; i<G.vexNum; i++)              // 对 shortestPath 数组元素进行初始化
      for(j=0; j<G.vexNum; j++)
         shortestPath[0][i][j]=G.arcs[i][j].adj ;
   for(k=1;k<=MAX;k++)
      for(i=0; i<G.vexNum; i++)           // 修正 shortestPath 数组元素的值
         for(j=0; j<G.vexNum; j++) {
            spk_1 = shortestPath[k-1][i][j];
            spk = min(|shortestPath[k-1][i][t]|+|(vt,vj)|) // t∈[1,|V|]
            shortestPath[k][i][j] = min( spk_1, spk ) ;
         }
}
```

3．Floyd 算法分析

以邻接矩阵为存储结构，Floyd 算法的时间复杂度为 $O(|V|^3)$，但处理的数据结构比较简单，比重复 |V| 次 Dijkstra 算法，使所有顶点依次为源点的求解方法实际时间花费要少。

Floyd 算法可处理权值为负的情况。

4.7 关键路径

由于现实世界的复杂性，现实生活的大部分工程都被划分为若干子工程，各子工程之间通常具有一定的相互制约，如某些子工程必须在另一些子工程完成之后才能开始。

对于工程，最重要和常受关注的问题有如下两个。

（1）完成整个工程的时间（工期）尽可能短。

（2）工程能否顺利进行，即各子工程之间的调度是否合理。

本节介绍第一个问题的解决办法：关键路径；第二个问题由图论的拓扑排序方法解决，在 4.8 节介绍。

4.7.1　关键路径概述

带权有向无环图中，若顶点表示事件，有向边表示工程的各项活动，有向边的权值表示活动的持续时间，称该带权无环有向图为边表示活动的网络，简称 AOE 网（Activity On Edges）网。AOE 网的典型应用为估算工程的工期以及给出缩短工期的办法，主要通过关键路径和关键活动求解。

图 4.24 为 AOE 网示例。其中：

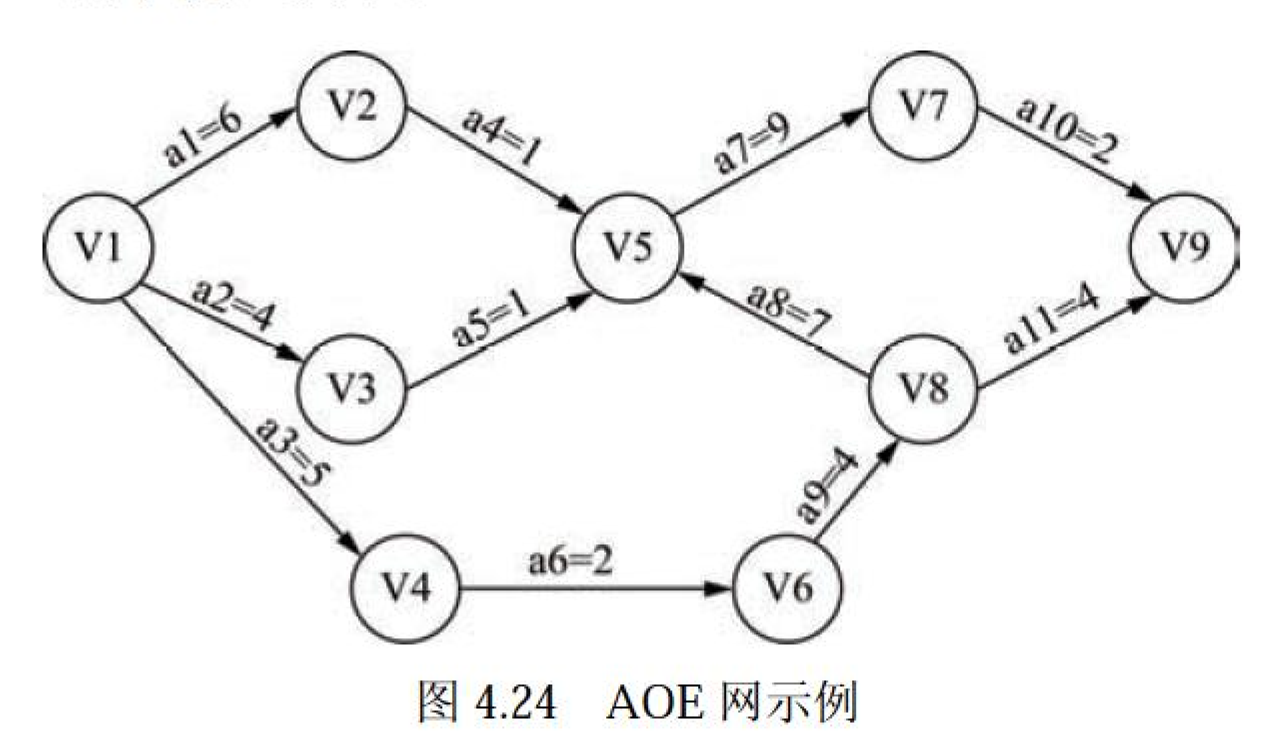


图 4.24　AOE 网示例

（1）事件 V1 表示整个活动的开始（源点），AOE 网有且仅有 1 个源点，是入度为 0 的顶点。

（2）事件 V9 表示整个活动的结束（汇点），AOE 网有且仅有 1 个汇点，是出度为 0 的顶点。

（3）工程从 V1 开始，至 V9 结束，中间历经 V2～V8 共 7 个事件、a1～a11 共 11 个活动。

（4）所有活动必须全部完成，所有事件必须全部发生，整个工程才结束。

（5）从源点 V1 到汇点 V9 有多条路径，但不是所有路径都对工期产生决定性影响。

（6）完成整个工程所需的时间取决于从源点到汇点的最长路径长度，即在这条路径上所有活动的持续时间之和，该路径称为关键路径。

（7）关键路径上的活动，称为关键活动。

4.7.2　关键路径求解

1. 约定

一般在实际工程应用中，希望工程的工期尽可能短。为缩短工期，必须知道哪些活动是影响工期的关键，即求解关键路径。为此进行如下约定。

（1）Ve(j)：事件 Vj 的最早发生时间，如 Ve(3)=4。

（2）Vl(j)：保证整个工期不推迟的情况下，事件 Vj 的最晚发生时间，如 Vl(3)=6。

（3）e(i)：活动 ai 的最早开始时间，如 e(5)=4。

（4）l(i)：保证整个工期不推迟的情况下，活动 ai 的最晚开始时间，如 l(5)=6。

（5）l(i)−e(i)：完成活动 ai 的时间余量，如 l(5)−e(5)=2。

（6）关键活动为 l(i)=e(i)的活动。

假设事件 Vi 和 Vj 之间的活动 ak 用弧 $<i, j>$ 表示，其持续时间用 $dut(<i, j>)$表示，则有：

（1）$e(k) = Ve(i)$。

（2）$l(k) = Vl(j) - dut(<i, j>)$。

2．求解流程

（1）从 $Ve(1) = 0$ 开始，利用如下公式向前递推：

$$Ve(i) = \max\{Ve(j) + dut(<j,i>)\},\quad <j,i> \in Hi,\quad j \in [2,|V|]$$

其中 Hi 是所有以事件 Vi 为头的弧的集合，|V|表示事件的个数。

（2）从 $Vl(|V|) = Ve(|V|)$ 利用如下公式向后递推：

$$Vl(i) = \min\{Vl(j) - dut(<i, j>)\},\quad <i, j> \in Ti,\quad i \in [1,|V|-1]$$

其中 Ti 是所有以事件 Vi 为尾的弧的集合，|V|表示事件的个数。

（3）根据 Ve(j)和 Vl(j)，可求得 e(i)和 l(i)，即可确定关键活动和关键路径。

3．求解示例

图 4.25 为图 4.24 的关键路径求解过程。

注意：

（1）关键路径未必唯一。

（2）并非加快任何一个关键活动都可缩短整个工程的工期。只有加快包括在所有关键路径上的关键活动才能缩短工期。

（3）关键活动的速度提高是受限的，提高过多可能导致原关键路径变为非关键路径。

（4）关键路径重在熟练掌握求解过程。

关键路径的求解过程：
求Ve，Vl；求e，l；
计算l-e

顶点	Ve	Vl
V1	0	0
V2	6	6
V3	4	6
V4	5	8
V5	7	7
V6	7	10
V7	16	16
V8	14	14
V9	18	18

活动	e	l	l-e
a1	0	0	0
a2	0	2	2
a3	0	3	3
a4	6	6	0
a5	4	6	2
a6	5	8	3
a7	7	7	0
a8	7	7	0
a9	7	10	3
a10	16	16	0
a11	14	14	0

图 4.25　关键路径求解示例

4.8 拓 扑 排 序

有向无环图表示某工程的各子工程及其相互制约时，用顶点表示活动，弧表示活动之间的先后顺序关系，称该图为顶点表示活动的网络，简称 AOV 网（Activity On Vertex Network）。AOV 网的典型应用为判断各活动之间的先后顺序关系是否合理，比如不能出现回路或环，即自己不能以自己为先决条件。

拓扑排序指无环 AOV 网将全部活动排列为线性序列，使得若 AOV 网存在弧 $<a_i, a_j>$，则在该线性序列中，a_i 一定排在 a_j 之前。

拓扑排序的方法如下。

（1）从 AOV 网中选择一个入度为 0 的顶点，并输出。

（2）从 AOV 网中删除该顶点及所有以其为尾的弧。

（3）重复（1）、（2），直至：

① 输出全部顶点，则按序输出的顶点序列即为拓扑排序结果。

② 当前图中不存在无前驱的顶点，说明该 AOV 网为有环网，无法进行拓扑排序。图 4.26 为拓扑排序示例。

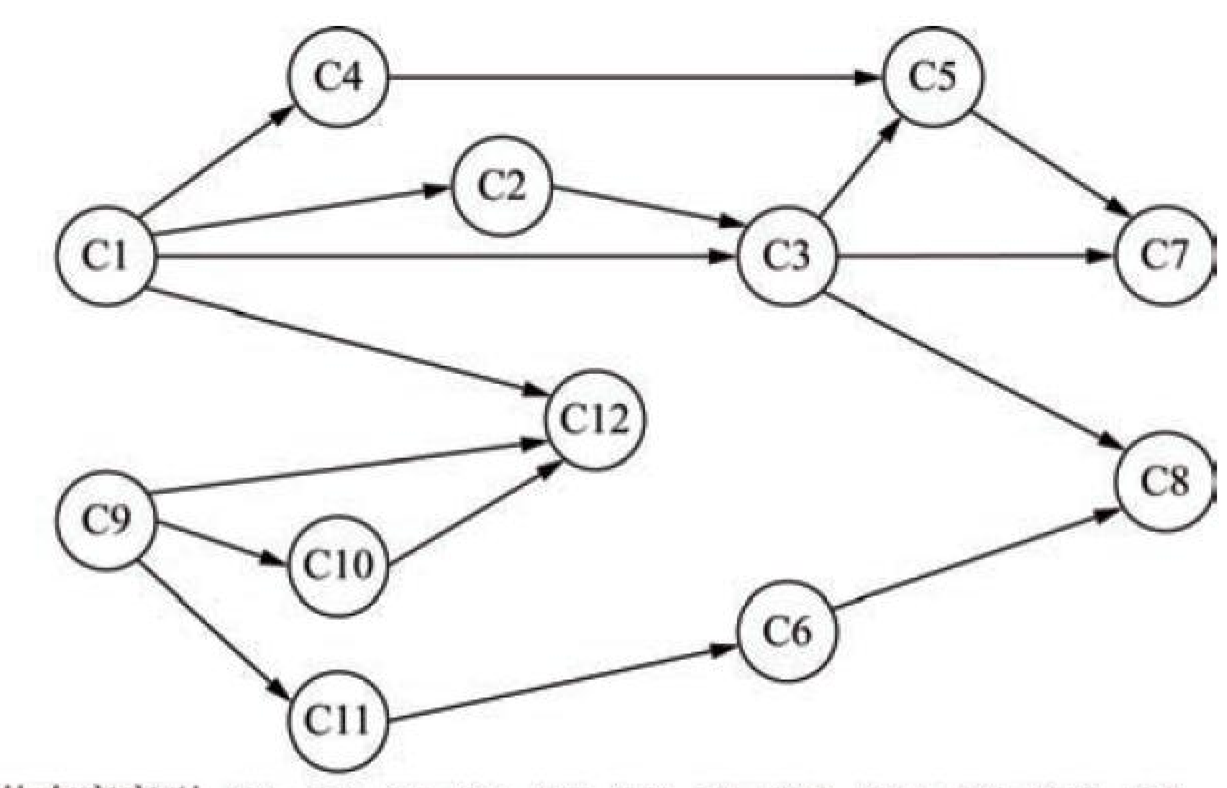


图 4.26　有向图的拓扑排序示例

注意：

（1）拓扑排序结果未必唯一。

（2）算法实现关键在于以下三点：

① 查找入度为 0 的顶点。

② 修正相关顶点的入度。

③ 每次输出任一入度为 0 的顶点。

（3）熟练掌握拓扑排序过程。

4.9 公共子表达式

有向无环图的主要应用，可节省含公共子表达式的表达式的存储空间。如：

$$[(x_1+x_2)(x_3-x_4)+(x_3-x_4)/x_5][(x_3-x_4)/x_5]$$

直接表示如图 4.27 所示。

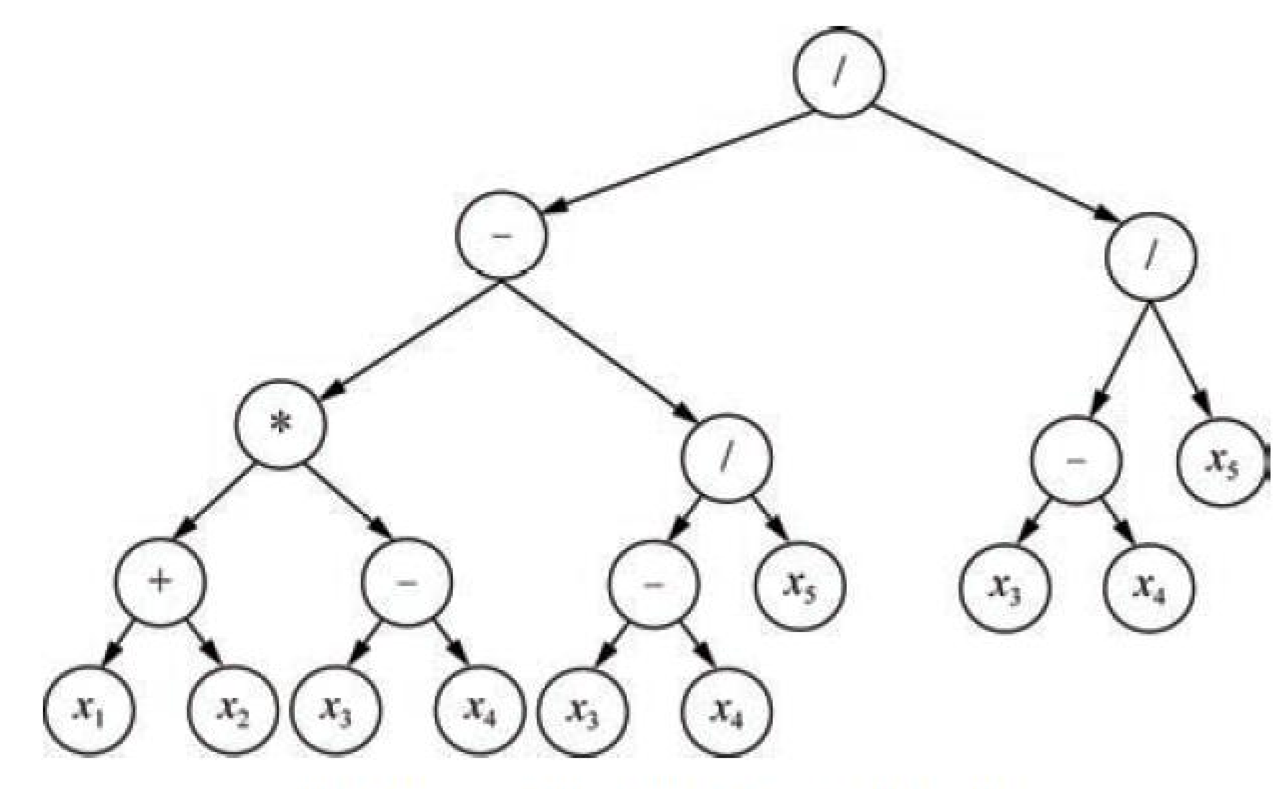


图 4.27 表达式的有向无环图示例

公共子表达式示例如图 4.28 所示。

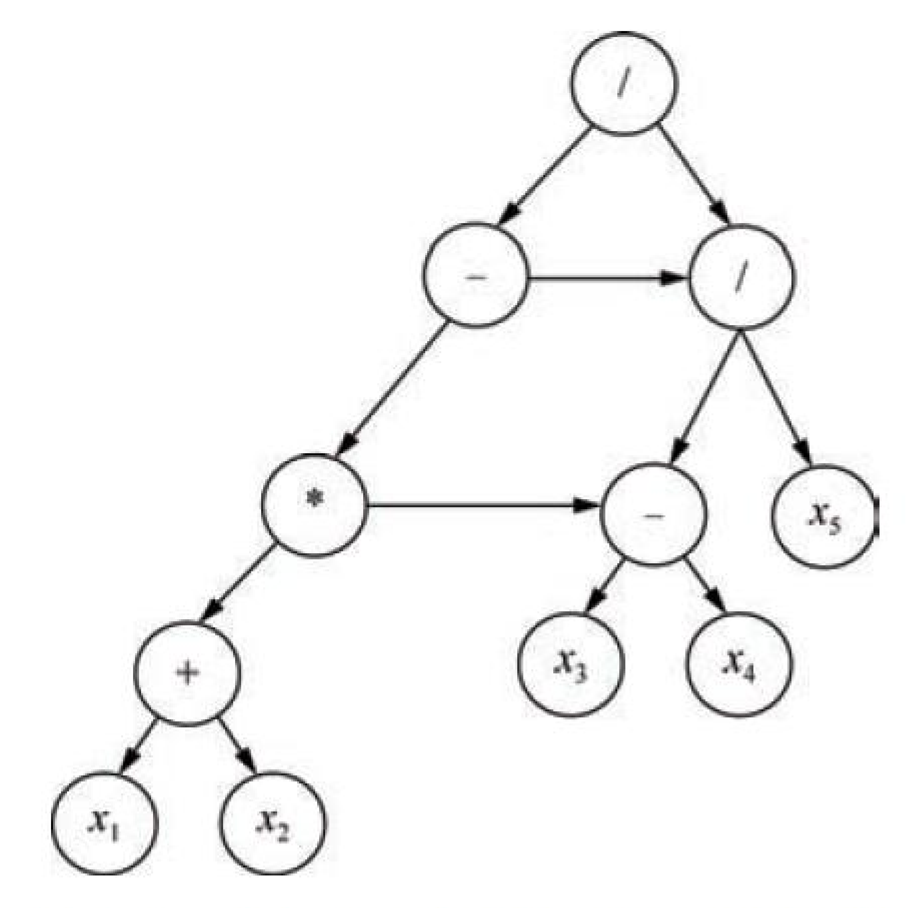


图 4.28 公共子表达式的有向无环图示例

注意：考生需要关注由公共子表达式的有向无环图写表达式和伪代码。

4.10 本 章 小 结

本章主要介绍图论的基础知识及常见应用，考生应重点理解基本概念及其伪码实现，掌握算法的本质内涵，为应对考题提供思路，也为灵活应用打好基础。

第 5 章　查找

本章学习目标

- 掌握静态查找的折半查找方法、散列查找方法及其性能分析。
- 掌握折半查找方法的条件（顺序存储、有序表）以及重点（查找过程中折半修改中间下标值 mid）。
- 掌握散列查找的散列函数构造及冲突解决办法。
- 掌握动态查找相关的二叉排序树的构造、二叉平衡书的调整。
- 重点理解 B 类树的概念、性质及基本操作。

5.1　本章导学


查找

5.1.1　知识结构

本章知识结构如图 5.1 所示，加粗框中的内容需要考生重点理解并掌握。

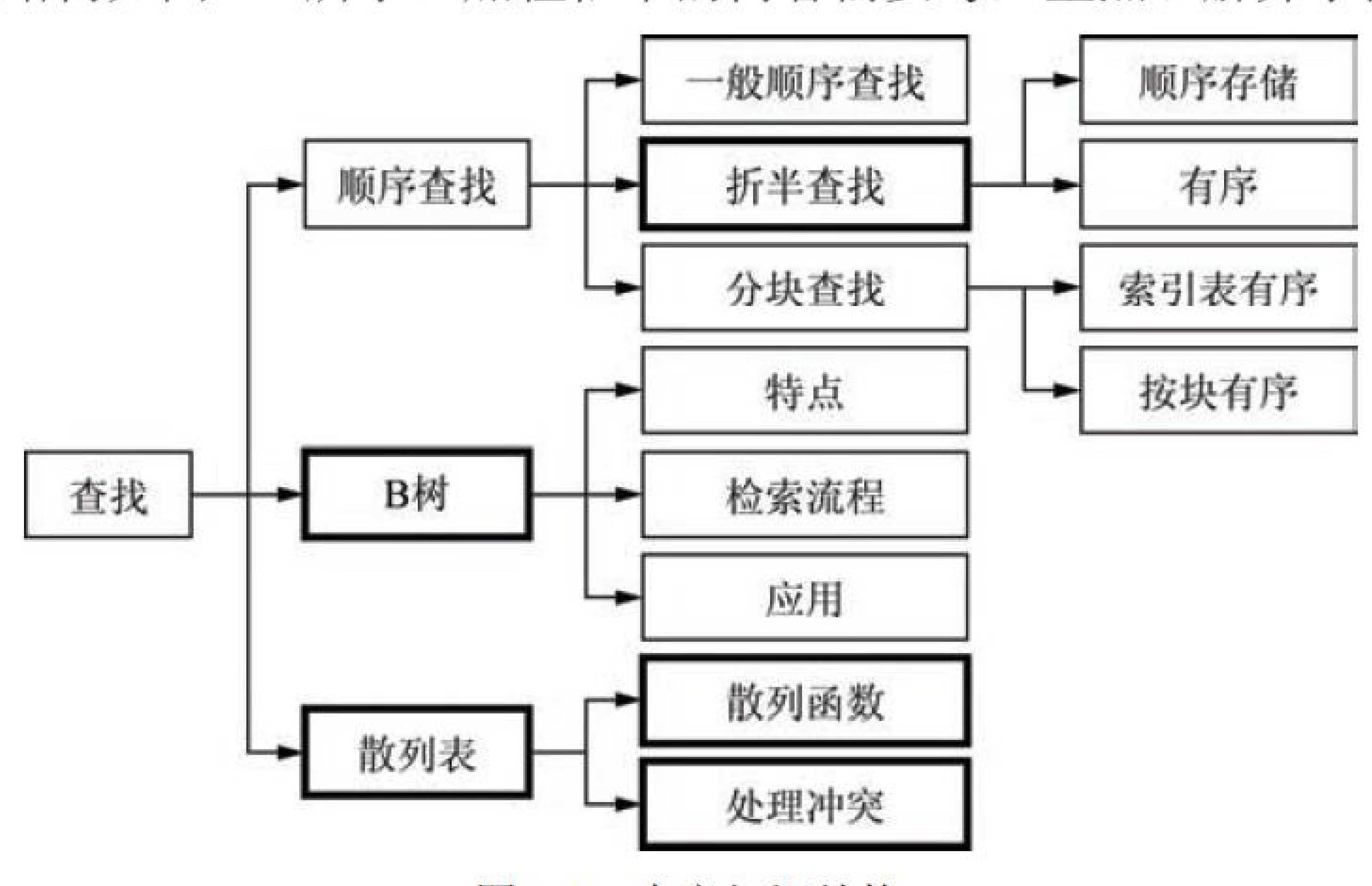


图 5.1　本章知识结构

5.1.2　命题特点

1. 命题规律

（1）本章是历年各高校硕士研究生招生考试的重点考查内容，考点较多。

（2）本章可单独命题，也可联合其他考点，比如和二叉树结合命题。

（3）查找、查找表的相关概念，静态查找表、动态查找表、散列表的基本概念，顺

序查找的过程及“岗哨”等，易出客观题。

（4）折半查找的过程、特征、算法实现比较重要，可灵活出题。

（5）B 树定义及构造过程可出客观题。

（6）散列表的构造过程、散列函数和处理冲突方法，易出主观题。

（7）查找效率易出客观题。

2．命题趋势

本章在全国硕士研究生入学考试初试中具有重要地位，关注散列表的主观题。

5.2 基 本 概 念

（1）查找表：同类型数据元素（或记录）构成的集合。由于“集合”中的数据元素之间的关系不受限制，因此“查找表”是一种非常灵活的数据结构。

（2）静态查找表：只能够查询某“特定”数据元素是否在查找表中，或检索某“特定”数据元素的各种属性的查找表为静态查找表。

（3）动态查找表：除查询、检索操作外，还可进行插入、删除数据元素的查找表，或查找表在查找过程中动态生成，查找成功，返回位置或记录信息；查找不成功，则插入待查关键字。

（4）关键字：数据元素或记录中能标识一个数据元素或记录的数据项的值。若关键字本身可以唯一标识一个记录，称为主关键字，否则称为次关键字。

（5）查找：又称为检索，指在查找表中确定一个其关键值字等于给定待查找关键值的记录或数据元素的过程。若表中存在这样的记录，则查找成功，可返回该记录信息，或指示该记录在查找表中的位置，否则为查找不成功。

（6）平均查找长度（ASL）：评价查找算法的指标。为给定值在查找表中的位置，需要和若干记录的关键字进行比较。其中，假设给定待查找关键值分别为所有记录关键字时所依次进行的比较次数的平均值称为查找成功时的平均查找长度。

5.3 顺序表的静态查找

顺序表的静态查找应用广泛，通常有顺序查找、折半查找、分块查找三种方式。

5.3.1 顺序查找

从查找表的第一个元素开始往后或从最后一个元素开始往前，顺序扫描线性表，依次比较待查找关键字值和数据元素的关键字是否相等。为提高查找效率，一般会浪费一

个数据元素空间（第一个或最后一个数据元素存储空间），即表长为 n 的线性表占用 n+1 个数据元素的存储空间，用来存放待查找关键字值，称为“岗哨”。

1．查找流程

（1）假设线性表存储于数组 a[n+1]。

（2）待查找关键字值 Key 放于 a[0]中：a[0]=Key。

（3）从数组元素 a[n]开始，向 a[0]方向依次比较 Key 和当前数组元素的关键字值，相等时返回数组元素下标。

（4）查找不成功时返回 0；查找成功返回和 Key 值相等的数组元素下标。

2．伪代码

```
#define maxSize 10000         // 顺序表的最大长度
typedef  struct {             // 定义结构体类型
   eleType Selem[maxSize];    // 线性表存储于数组 Selem，存储数据从下标 1 开始
   unsigned  length;          // 顺序表的当前长度
}Slist;                       // 静态分配空间的顺序表的类型名为 Slist
unsigned  int  seqSearch(Slist  a,  eleType  Key){
   a.Selem[0]=Key;            // 岗哨
   for(i=a.length; a.Selem[i]!=Key;i--)
;
return i;
}
```

3．算法分析

设线性表包含 n 个元素，每个元素查找成功的概率相等，$p_i=1/n$，则：

（1）查找不成功：与 a[n]～a[0] 依次进行比较，比较次数为 n+1，即 $ASL_{不成功}=n+1$。

（2）查找成功：$ASL_{成功}=\sum(n-i+1)/n=(n+1)/2$。

（3）时间复杂度为 O(n)。

顺序查找的优点是：对存储结构无要求；对数据元素的顺序无要求。缺点是平均查找长度较大；查找效率低，n 较大时，不易采用。

5.3.2　折半查找

折半查找又称为二分查找，当线性表采用顺序存储结构，且表中的元素有序时（升序或降序均可，本节以升序为例），不必依次查找数组元素，可快速确定待查找关键字值的区域，加快查找速度，提高查找效率。

1．查找流程

待查找关键字值和线性表中间位置的元素进行比较：

（1）相等：查找成功，返回下标。

（2）小于：在线性表的前半部分继续折半查找。

（3）大于：在线性表的后半部分继续折半查找。

（4）中间位置不存在时查找失败，返回 –1。

2. 伪代码

```
#define maxSize 10000          // 顺序表的最大长度
typedef struct {               // 定义结构体类型
   eleType Selem[maxSize];     // 线性表存储于数组Selem中，存储数据从下标0开始
   unsigned length;            // 顺序表的当前长度
}Slist;                        // 静态分配空间的顺序表的类型名为Slist
int halfSearch(Slist a, eleType Key){
   low=0;
   high=n-1;
   mid=(low+high)/2;
   while(low<=high){
       if(a[mid]==Key)
          return mid;
       if(a[mid]>Key)
          high=mid-1;
       else
          low=mid+1;
   }
   return -1;
}
```

3. 示例

假设有序线性表存储于数组 a[10]={6,8,19,20,32,59,66,82,125,258}，Key=19，折半查找过程如图 5.2 所示。

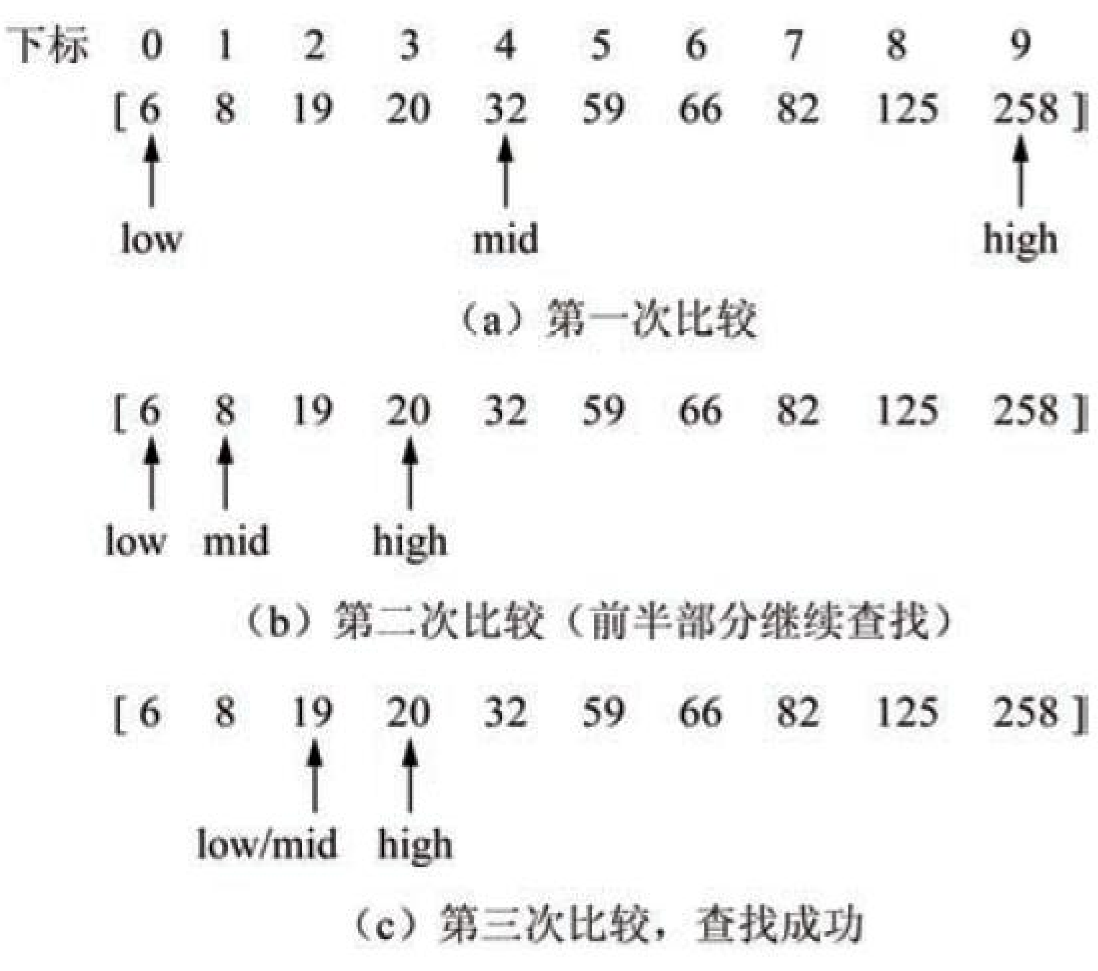


图 5.2 有序顺序表折半查找过程示例

图 5.3 为折半查找相应的判定树。

共经过 3 次比较，成功查找到“19”。折半查找为二叉判定树的遍历过程，对其进行中序遍历，即可得到已知的有序序列。

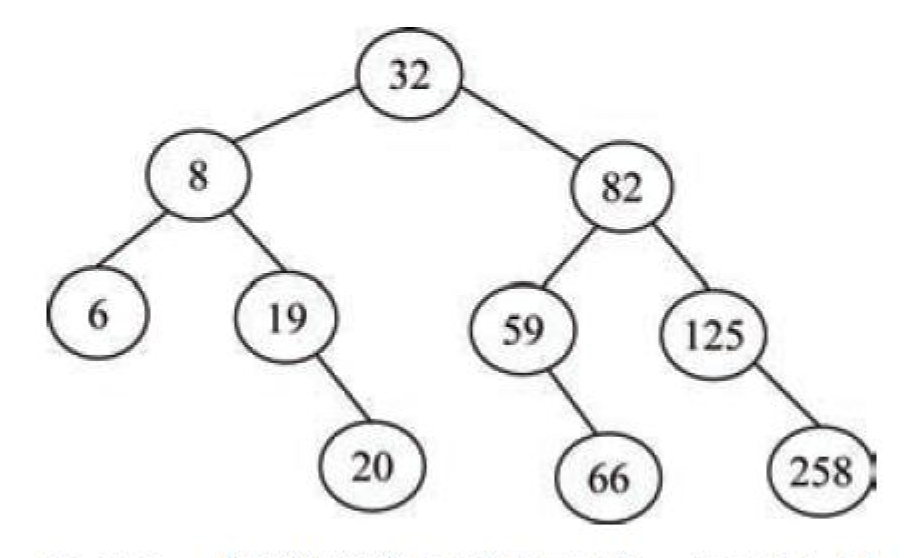


图 5.3 有序顺序表折半查找二叉判定树

4. 算法分析

设有序顺序表包含 $n=2^h-1$ 个数据元素，h 为二叉判定树的高度，每个元素查找成功的概率相

等，即 $p_i=1/n$，则：

（1）查找成功进行的比较次数为该数据元素在二叉判定树中的层数。

（2）最多进行 h 次比较。

（3）查找不成功：比较次数为判定树的深度，即 $ASL_{不成功}=\lceil \log_2(n+1) \rceil$。

（4）查找成功：$ASL_{成功}=\sum(i \times 2^i)/n=\lceil \log_2(n-1) \rceil$。

（5）时间复杂度为 $O(\log_2 n)$。

折半查找的优点是效率高。缺点是必须顺序存储，而且线性表中的关键字必须有序。

5.3.3　分块查找

分块查找是顺序查找和折半查找相结合的方法，又称为索引顺序查找，由索引表和查找表两部分组成，要求索引表有序，查找表分块，且块内无序、块间有序。

查找表部分：部分有序。将查找表划分为若干块，且第 i 块的最大关键字值小于第 i+1 块的最小关键字值，块内无要求。

索引表部分：有序，为结构体数组，每个数组元素包含两个成员：每块的最大关键字和该块第一个数据元素在查找表中的位置。如图 5.4 所示，其中上半部分为索引表，下半部分为查找表。

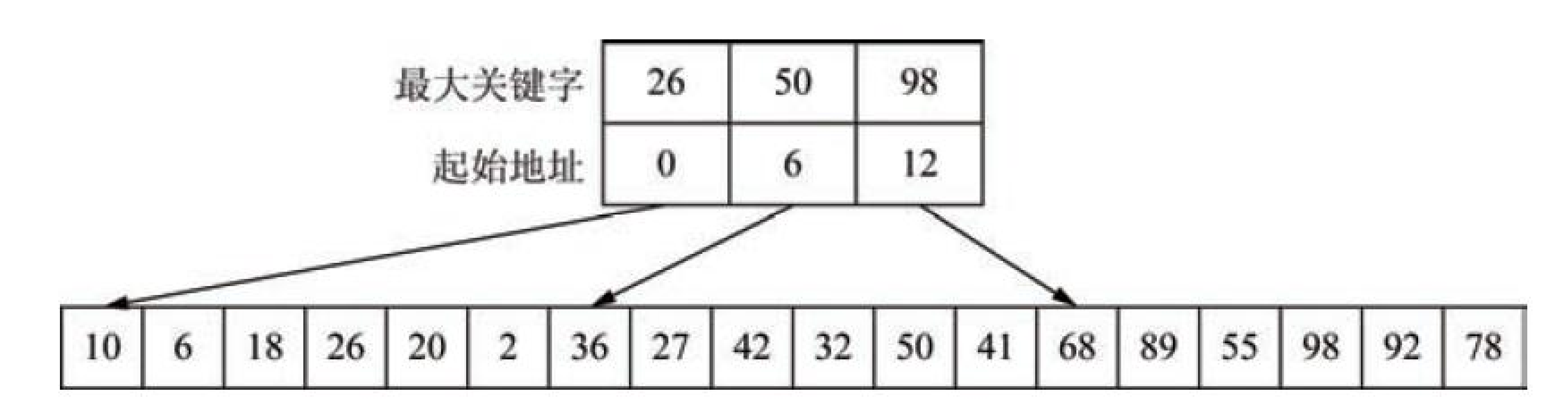


图 5.4　分块查找的索引表及查找表

1．查找流程

（1）查找索引表：确定待查关键字值在查找表中的块位置。

（2）检索查找表：确定待查关键字值在查找表中的具体位置，或查找不成功。

2．示例

图 5.4 查找关键字 92 的过程如下。

（1）查找索引表：92 分别和 50、98 进行比较，确定 92 在查找表中位于第 12 个数据元素开始的块。

（2）检索查找表：92 和 68、89、55、98、92 进行 5 次比较，得出其在查找表中的位置为 12+(5−1)=16。

3．算法分析

设索引表包含 n_{index} 个数据元素，查找表包含 n 个数据元素，共 k 块，每块包含 n_i 个数据元素，每个元素查找成功的概率相等，即 $p_i=1/n$，则：

（1）分块查找的平均查找长度 $ASL_{分块}$为索引表的平均查找长度 ASL_{index} 与查找表的平均查找长度 ASL_{seqi} 之和。

（2）索引表可进行折半查找，$ASL_{index}=\lceil \log_2(n_{index}+1) \rceil$。

（3）查找块需要顺序查找，$ASL_{seqi}=n_i+1$。

（4）若每块包含元素个数相等，且索引表和查找表均顺序查找，则

$$ASL_{分块} = ((n/k)^2+2*n/k+n) / (2*n/k)$$

（5）当 $n/k=\sqrt{n}$ 时，索引表和查找表均顺序查找时 $ASL_{分块}=\sqrt{n}+1$；索引表折半查找、查找表顺序查找时 $ASL_{分块}=\lceil \log_2(k+1) \rceil + n/k/2$。

分块查找结合顺序查找和折半查找的优点，效率介于二者之间。对分块查找来说，索引表最好顺序存储是按关键字有序。

5.4 二叉排序树

二叉排序树属于动态查找表，基本概念和算法见 3.3.6 节，本节主要介绍二叉排序树在查找中的应用。

1．二叉排序树的查找流程（假设二叉排序树的中序序列为升序）

（1）二叉排序树为空，返回查找失败信息。

（2）待查找关键字值和根结点的关键字值进行比较，若

- 相等：则查找成功，返回根结点。
- 小于：则继续在根结点的左子树上查找。
- 大于：则继续在根结点的右子树上查找。

（3）重复（2），直到树空，返回查找失败信息。

2．伪代码

```
BiTree BSTSearch ( BiTree  T,  eleType  key ) {
    if( (!T)
       return  -1;                        // 没找到，查找失败
    if( (T) && key == T->data.key )
       return  T;
    if( key < T->data.key )
       return  BSTSearch (T->lChild,key);
    else
       return  BSTSearch (T->rChild,key);
}
```

3．性能分析

假设二叉排序树有 n 个结点，则有如下特点。

（1）二叉排序树的深度范围为$[\log_2 n, n]$（有序序列构造的二叉排序树为单左子树或单右子树，深度为 n）。

（2）二叉排序树查找成功的比较次数不会超过二叉排序树的深度。

（3）二叉排序树查找最好情况下的时间复杂度为 $O(\log_2 n)$，最坏情况下的时间复杂度为 O(n)。

5.5　二叉平衡树

基本概念和算法见 3.3.7 节，由于二叉排序树可能出现最高树（单左子树或单右子树），导致其查找的时间复杂度达 O(n)，查找效率低下。二叉平衡树为特殊二叉排序树，任意结点的平衡因子绝对值不超过 1，故可有效降低查找的时间复杂度，为 $O(\log_2 n)$。

假设以 N_h 表示高度为 h 的平衡二叉树中含有的最少结点数，则有：

- $N_0=0$。
- $N_1=1$。
- $N_h=N_{h-1}+N_{h-2}+1$。

最多结点数为高度 h 的满二叉树的结点数：$N_{max}=2^h-1$。

注意：

（1）二叉排序树的查找长度取决于树的形态，最次形态为单枝树。

（2）二叉平衡树（AVL 树）的查找长度取决于树的高度，由于平衡因子绝对值不超过 1，故树的高度较小。

（3）二叉排序树和二叉平衡树的最好形态为满二叉树（包含更多结点的最小高度树）。

（4）一般顺序查找、折半查找、分块查找、二叉排序树查找、二叉平衡树查找每个节点采访一个记录，不适合单结点含大量信息的应用。

5.6　B 树 类

5.6.1　B 树

B 树的每个结点可存放若干记录，多用于文件系统。

1．定义

B 树是一种平衡多路查找树。一棵 m 阶的 B 树，或为空树或为满足以下特性的 m 叉树：

（1）树中每个结点至多含 m 棵子树。

（2）若根结点不是叶子结点，则至少含两棵子树。

（3）除根结点外的所有非终端结点至少含 $\lceil m/2 \rceil$ 棵子树。

（4）所有的非终端结点的结构为（$n, A_0, K_1, A_1, K_2, A_2,\cdots, K_n, A_n$），其中，$K_i$ ($i = 1,2,\cdots, n$) 为升序排列的关键字；A_i ($i = 0,1,\cdots, n$) 为指向子树根结点的指针，且 A_i 所指子树中所有结点的关键字均小于 k_i，A_n 所指子树中所有结点的关键字均大于 k_n，n（对于根结点 $1 \leqslant n \leqslant m-1$，对于其他结点 $\lceil m/2 \rceil - 1 \leqslant n \leqslant m-1$）为关键字的个数（或 $n+1$ 为子树个数）。

（5）所有叶子结点在同一个层次上，且不含有任何信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

图 5.5 是一棵 4 阶 B 树，最后一层的长条框表示叶子结点。

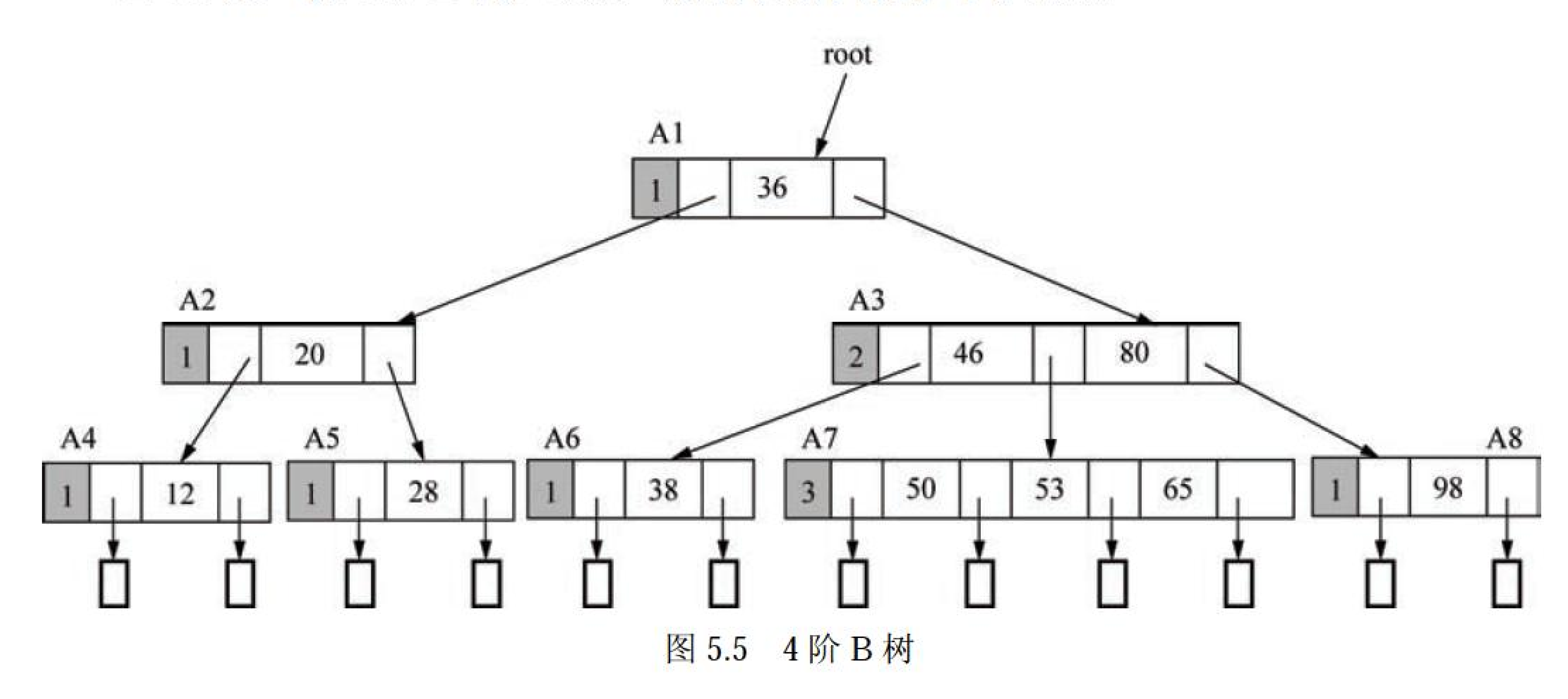


图 5.5　4 阶 B 树

2．B 树查找

在 B 树上进行查找的过程与二叉排序树查找类似，区别如下。

（1）B 树每个结点为多关键字有序表，在到达某个结点时，先在有序表中查找，若找到，则查找成功；否则到按照对应的指针信息指向的子树中去查找。

（2）到达叶子结点时，说明树中没有对应的关键码，查找失败。

1）图 5.4 的 B 树查找成功过程如下（找 65）

（1）从根 root 指向的结点 A1 开始，65>36，则沿 36 右边的指针往下在结点 A3 查找。

（2）结点 A3 为含 2 个关键字的有序表，二分查找确定 65 所在结点的方向或位置。

① low=0，high=1，low<=high，mid=(0+1)/2=0，65 大于第 0 个关键字 46，继续二分查找。

② low=mid+1=0+1=1，low<=high，mid=(1+1)/2=1，65 小于第 1 个关键字 80，故沿 46 和 80 之间的指针往下在结点 A8 查找。

（3）结点 A8 为含 3 个关键字的有序表，二分查找确定 65 所在结点的方向或位置。

① low=0，high=2，　mid=(0+2)/2=1，low<=high，65 大于第 1 个关键字 53，继续二分查找。

② low=mid+1=1+1=2，low<=high，mid=(2+2)/2=2，65 等于第 2 个关键字，查找成功。

2）图 5.4 的 B 树查找不成功过程如下（找 66）

（1）从根结点 AI 开始，66>36，则沿 36 右边的指针往下在结点 A3 查找。

（2）结点 A3 为含 2 个关键字的有序表，二分查找确定 66 所在结点的方向或位置。

① low=0， high=1，low<=high， mid=(0+1)/2=0，66 大于第 0 个关键字 46，继续二分查找。

② low=mid+1=0+1=1，low<=high，mid=(1+1)/2=1，66 小于第 1 个关键字 80，故沿 46 和 80 之间的指针往下在结点 A8 查找。

（3）结点 A8 为含 3 个关键字的有序表，二分查找确定 66 所在结点的方向或位置。

① low=0，high=2，low<=high，mid=(0+2)/2=1，66 大于第 1 个关键字 53，继续二分查找。

② low=mid+1=1+1=2，low<=high，mid=(2+2)/2=2，66 大于第 2 个关键字 65，继续二分查找。

③ low=mid+1=2+1=3，low>high，查找失败。

3）B 树查找性能分析

B 树查找包含两步。

（1）B 树找结点：在磁盘文件中按关键字所属范围指针查找待查关键字所属结点，并将找到的结点读入内存。

（2）结点内找关键字：二分或顺序查找确定待查关键字的位置或所属结点，操作在内存中进行。

（3）重复上述两步，直到查找成功，或找到叶子结点，查找失败。

由于磁盘速度远低于内存速度，所以 B 树找结点是决定 B 树查找效率的关键。假设各关键字查找概率相等，则 B 树深度越大，效率越低。

由定义可知，深度 h 的 m 阶 B 树：

（1）第一层至少有 1 个结点。

（2）第二层至少有 2 个结点。

（3）第三层至少有 $2\times(\lceil m/2\rceil)$（除根之外的每个非终端结点至少有 $\lceil m/2\rceil$ 棵子树）。

（4）第 k 层至少有 $2(\lceil m/2\rceil)^{k-2}$ 个结点。

（5）第 h 层全为叶子结点，若深度 h 的 m 阶 B 树中具有 n 个关键字，则叶子结点即查找不成功的结点为 n+1，有：

$$n+1\leqslant 2\times\lceil m/2\rceil^{h-2} \longrightarrow h\leqslant \log_{\lceil m/2\rceil}((n+1)/2)+2$$

即在含有 n 个关键字深度 h 的 m 阶 B 树上进行查找时，从根结点到关键字所在结点的路径上涉及的结点数不超过 $\log_{\lceil m/2\rceil}((n+1)/2)+2$。

3．B 树插入关键字（构造 B 树）

插入关键字的流程如下。

B 树的生长方向为自下向上（根）。

（1）空树直接生成结点，填入关键字。

（2）在 m 阶非空 B 树上插入关键字，即在最底层的某非终端结点添加一个关键字：

① 该结点的关键字个数不超过 m−1 个，直接添加，插入成功。

② 该结点的关键字已达 m 个，添加后分裂该结点。

- 结点分成 3 部分。
- 第 1 和第 3 两部分关键字个数为$\lceil m/2 \rceil$−1。
- 第 2 部分为含一个关键字的结点，第 1、3 部分分别作为第 2 部分结点的左右子树。
- 第 2 部分结点插入其父结点。
- 若其父亲结点的当前关键字为 m 个，同样的方法进行分裂，直到某父结点的关键字个数小于 m 为止。

假设关键字序列为{20,54,69,84,71,30,78,25,93,1,7,76,51,66,68,53, 3,79,35,12,15,6}，则构建 5 阶 B 树的过程如图 5.6 所示（简化图，不画叶子）。

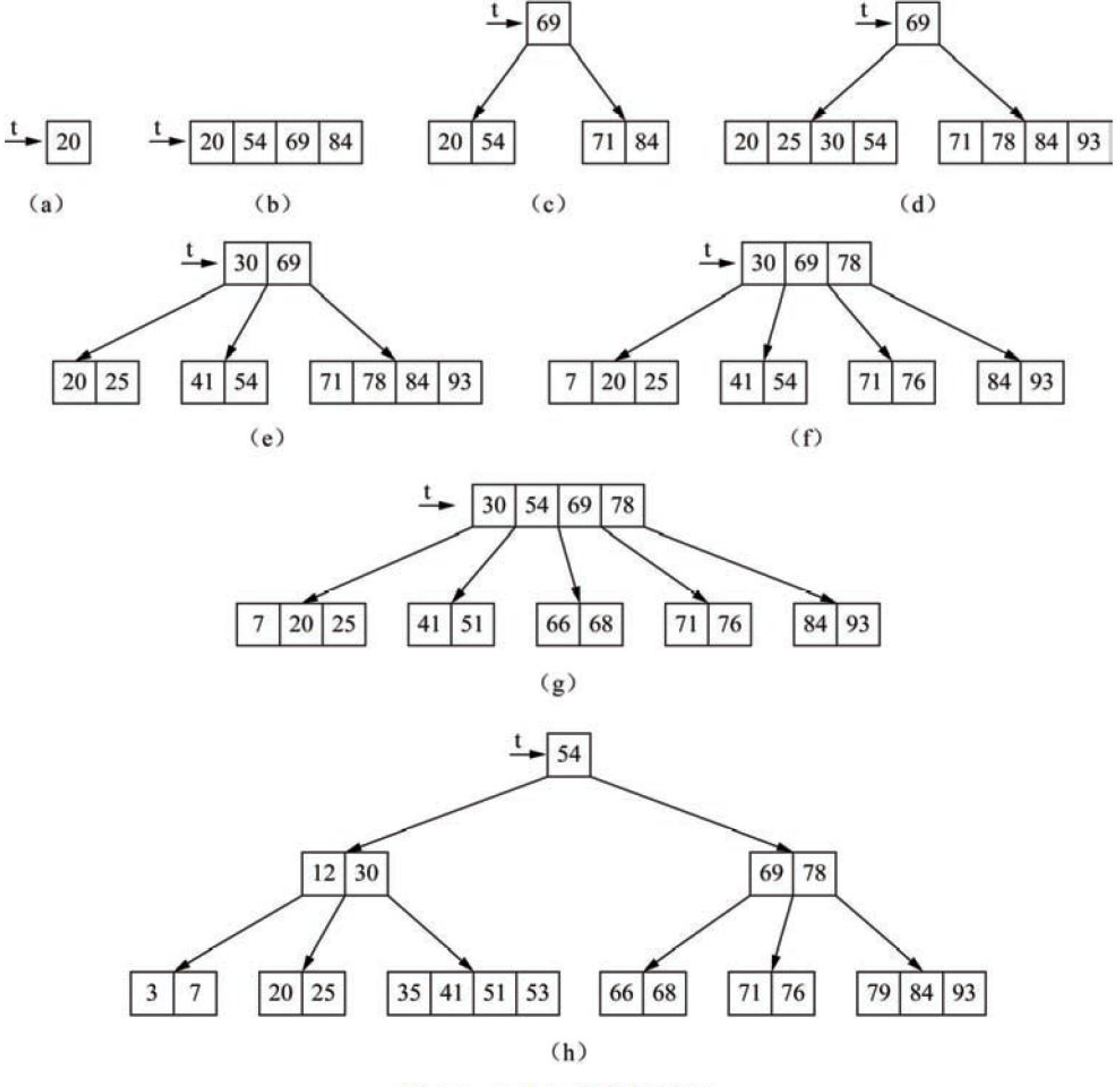


图 5.6　5 阶 B 树构造过程

（1）空树中插入 20，见图 5.6（a）。

（2）依次插入 54，69，84，见图 5.6（b）。

（3）插入 71，关键字个数达到 5，分裂成三部分：{20,54}，{69}，{71,84}，并将 69 置为其他两部分的父结点，见图 5. 6（c）。

（4）依次插入 30，78，25，93，见图 5.6（d）。

（5）插入 41，关键字个数达到 5，分裂成三部分：{20,25}，{30}，{41,54}，并将 30 插入其父结点，父结点的前两个指针分别指向其余两部分，见图 5. 6（e）。

（6）7 直接插入；76 插入后，关键字个数达到 5，分裂成三部分：{71,76}，{78}，{84,93}，并将 78 插入其父结点，父结点的后两个指针分别指向其余两部分，见图 5.6(f)。

（7）51，66 直接插入；68 插入后，关键字个数达到 5，分裂成三部分：{41,51}，{54}，{66,68}，并将 54 插入其父结点，父结点的第 2、3 指针分别指向其余两部分，见图 5.6（g）。

（8）直接插入 53，3，79，35；12 插入后，关键字个数达到 5（关键字序列 3，7，12，20，25），分裂成三部分：{3，7}，{12}，{20，25}，并将 12 插入其父结点；父结点关键字个数达到 5（关键字序列 12，30，54，69，78），分裂成三部分：{12，30}，{54}，{69，78}，54 为根结点，其左右子树分别指向 {12，30} 和 {69，78}；{12，30}三个指针分别指向{3，7}，{20，25}和 {35，41，51，53}；{69，78}三个指针分别指向{66，68}，{71，76}和 {79，84，93}，见图 5. 6（h）。

4．B 树删除关键字

B 树删除的流程如下。

（1）删除非最下层的、非终端结点 A_i 中的关键字（假设删除的关键字值为 K_j）。

①用 A_i 所指右子树的最小关键字 min（位于子树最下层非终端结点）代替 A_i 中的 K_j（i≥0，j≥1）。

② min 原结点中删除关键字 min，转到（2）。

③ 删除图 5.6（h）中的关键字 54 示例：用 54 右子树的最小关键字 66 替代 54，在原 66 所在的最下层非终端结点中删除 66。

（2）删除最下层非终端结点 B_m 中的关键字（假设删除的关键字值为 K_n）：

① 如果 B_m 中的关键字数目不少于 $\lceil m/2 \rceil$，直接删除 K_n 及其后的空指针即可。图 5.6（h）中删除关键字 79 之后的结果如图 5.7（a）所示。

② 如果 B_m 中的关键字数目少于 $\lceil m/2 \rceil$，需要合并结点：

- K_n 所在结点中的关键字个数为 $\lceil m/2 \rceil-1$，且与该结点相邻的左或右兄弟结点中的关键字个数大于 $\lceil m/2 \rceil-1$：将其兄弟结点中的最大（左兄弟结点）或最小关键字（右兄弟结点）上移至双亲结点，同时将双亲结点中大于或小于且和该上移关键字相邻的关键字下移至被删除关键字所在结点。图 5.6（h）中删除关键字 76 之后如图 5.7（b）所示。

- K_n 所在结点及其相邻的兄弟结点中的关键字个数均为⌈m/2⌉-1（最小合法值）。假设该结点有右兄弟，且其右兄弟结点地址由双亲结点中指针 B_{mk} 指示，则删除关键字之后，K_n 所在结点的剩余关键字和指针、双亲结点中的关键字一起合并到 B_{mk}（如果没有右兄弟，则合并到左兄弟结点，过程类似）。若操作使双亲结点中的关键字个数小于⌈m/2⌉-1，同样处理。图 5.6（h）中删除关键字 76 之后如图 5.7（c）所示。

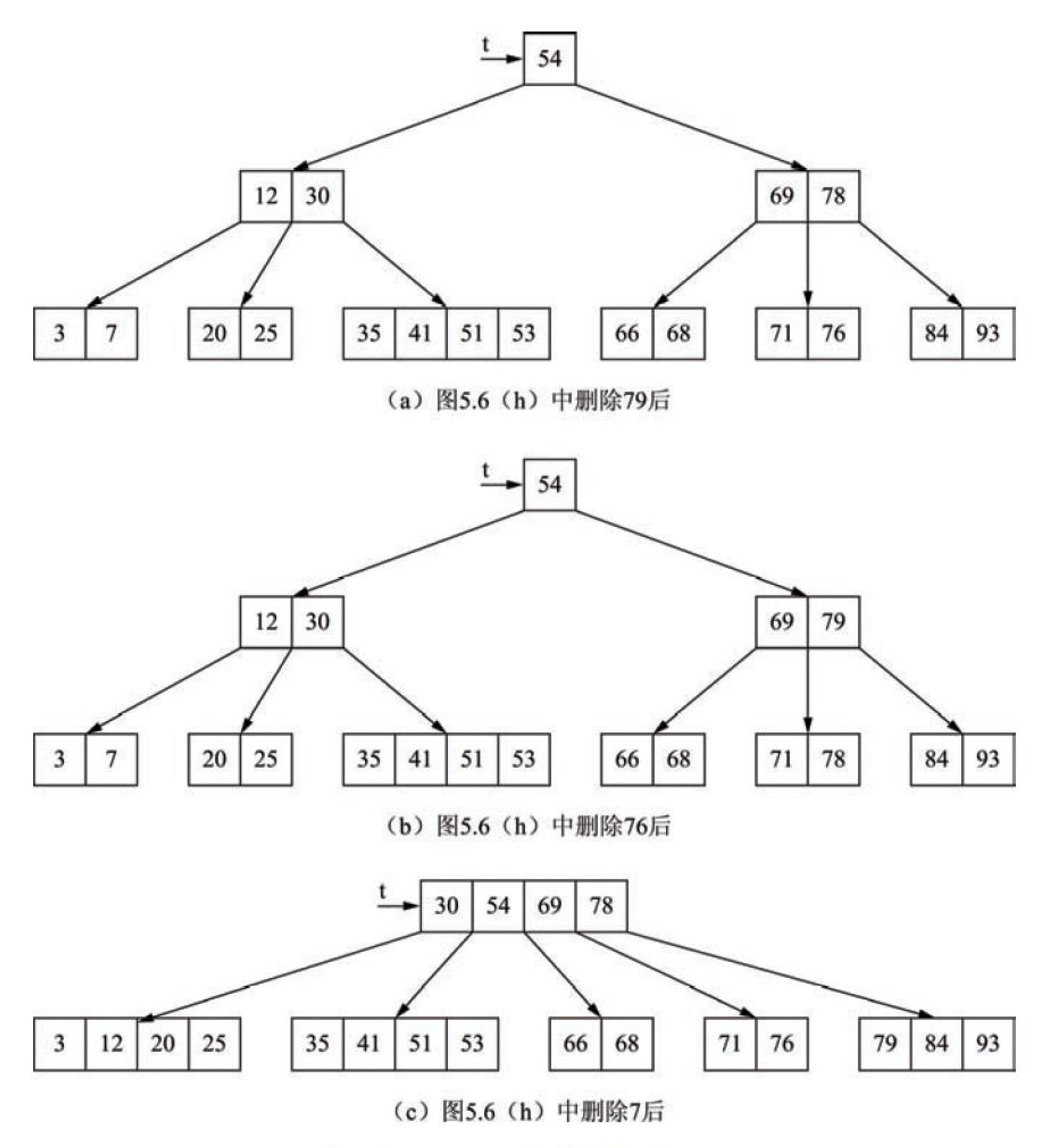


图 5.7　5 阶 B 树的删除结点示例

5.6.2　B+树

B+树是 B 树的变型。m 阶 B+树与 m 阶 B 树的区别有如下几点。

（1）有 n 棵子树的结点中包含 n 个关键字。

（2）每个结点的关键字个数和孩子指针个数相等。

（3）所有叶子结点中包含了全部关键字信息，以及指向包含这些关键字的记录的指针。

（4）叶子结点按关键字升序链接，构成单链表。

（5）所有非终端结点可以看作索引部分，结点中仅含其子树（根结点）中的最大或最小关键字。

（6）通常在 B+树上有两个头指针，一个指向根结点；一个指向关键字值最小的叶子结点。

（7）B+树进行两种查找运算，一种是从最小关键字开始，顺序查找；另一种是从根结点开始，随机查找。

（8）B+树上进行随机插入、删除和查找的过程基本与 B 树相应操作类似，不同的是在查找时，若非终端结点上的关键字等于待查找关键字值，查找仍继续进行到叶子结点，即查找路径从根结点到叶子结点。

图 5.8 是一棵 3 阶 B+树。其中 root 指向根结点，sqt 指向关键字值最小的叶子结点。

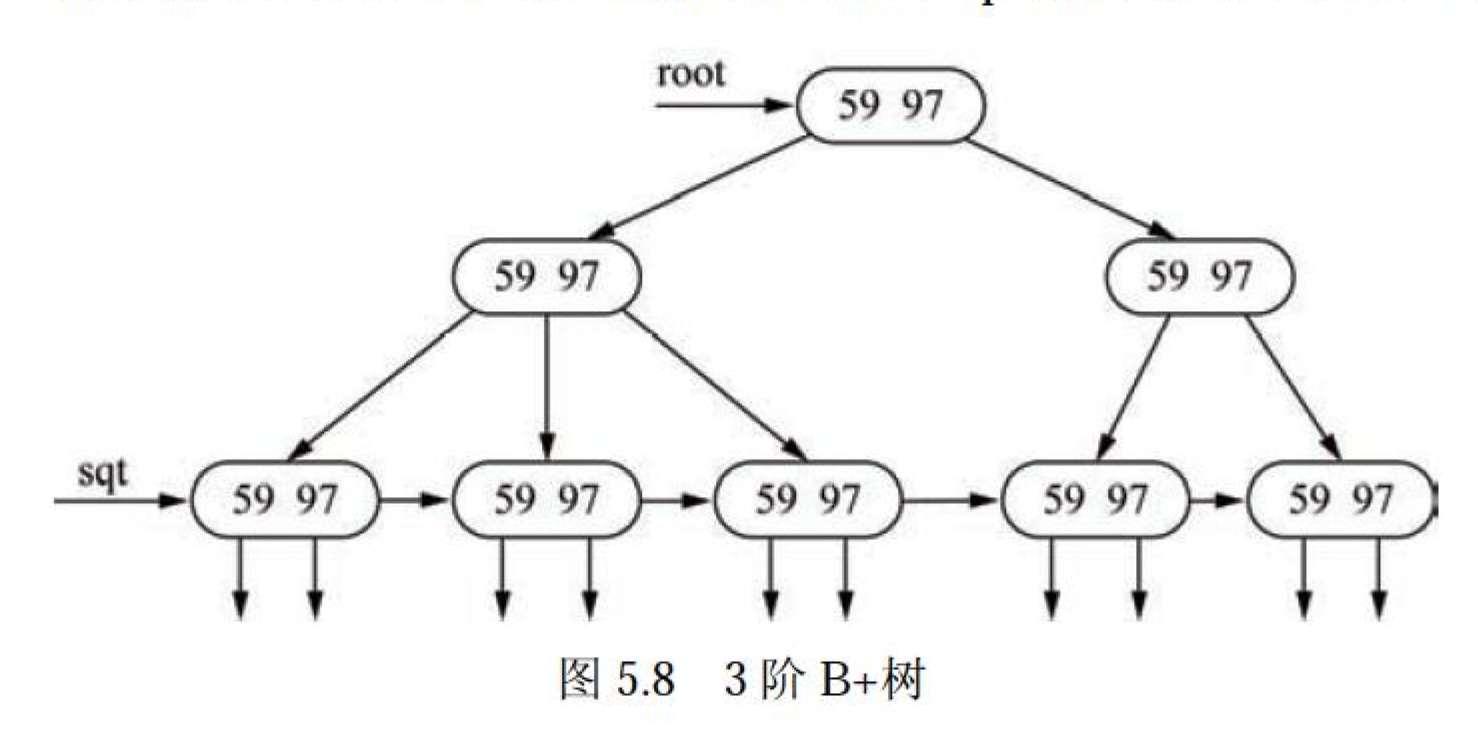


图 5.8　3 阶 B+树

5.7　散　列　表

上述查找过程均基于比较，故查找算法的时间复杂度取决于比较的次数。如果记录在查找表中的位置和其关键字值有关，则可直接由关键字值求出存储位置，无须比较。散列表即为具有此性质的线性表（查找表）。

5.7.1　基本概念

1．散列函数

散列函数又称哈希函数，以关键字值为自变量、函数值为记录在表中的存储位置的函数。如以 key 为关键字值， H(x)为散列函数，则 H(key)为 key 所在记录位于查找表中的位置。

2．散列地址

由散列函数得到的记录在查找表中的存储位置，有可能发生冲突。

3．冲突

key1≠key2，但 H(key1)=H(key2)，即 key1 和 key2 对应到同一存储位置，称发生了冲突。

4．同义词

$key_1 \neq key_2$，但 $H(key_1)=H(key_2)$，称 key_1 和 key_2 为同义词。

5．散列表

由散列函数和冲突处理方法将一组关键字映射到一个连续、有限的地址区间，该地址区间对应的查找表称为散列表。

5.7.2 散列函数构造

1．构造规则

（1）计算简单：计算时间不应超过上述查找算法的比较时间。

（2）分布均匀：关键字的散列函数值尽可能分布均匀，减少冲突。

2．常见散列函数

（1）除留余数法：最常见的散列函数

H (key)=key % m；（m 为正整数）

特点：m 的选取很重要，若散列表表长为 n，则要求 m≤n，且接近 n。m 一般为素数。

（2）直接定址法

H(key)=a*key+b；（a、b 均为常数）

特点：关键字基本连续时，该散列函数较有效。

（3）平方取中法

取关键字值平方的中间若干位作为散列地址。

（4）数值分析法

假设关键字由 k 位组成，每位可能有 r 个不同的数码（r 进制）。根据数码在各位的分布情况，选取数码出现频率大致相同的某几位作为散列地址。

特点：关键字改变，需重新分析。

（5）折叠法

将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为散列地址。

特点：关键字位数较多，且每位数码分布大致均匀时，可采用该散列函数。

（6）随机数法

取关键字的随机函数值为其散列地址，该方法较少使用。

5.7.3　处理冲突方法

常见的处理冲突方法如下。

1．开放定址法

常用处理方法为

$$H_i=(H(key)+d_i) \bmod n \qquad i=1,2,3,\cdots,k \quad (k\leqslant n-1)$$

其中，H(key)为散列函数；n 为散列表长；d_i 为增量序列。由 d_i 的不同分为以下三种不同的散列方式。

（1）线性探测再散列：$d_i=1,2,3,\cdots,n-1$。

示例：将一组关键字{19,23,1,68,20,84,55,11,10,79}，按照散列函数 H(key)=key % 11 和线性探测再散列处理冲突求解散列表。

解：求解各关键字的散列函数值

H(19)= 19 % 11=8　　H(23)= 23 % 11=1　　H(1)= 1 % 11=1　　H(68)= 68 % 11=2

H(20)= 20 % 11=9　　H(84)= 84 % 11=7　　H(55)=55 % 11=0　　H(11)= 11 % 11=0

H(10)= 10 % 11=10　　H(79)= 79 % 11=2

各关键字的散列函数值和存放位置如下。

① 19、23：直接放入其对应散列地址 8 和 1。

② 1：和 23 为同义词，地址 1 发生冲突，放入 H(H(1)+1)= H(2)= 2 % 11=2。

③ 68：散列地址 2 已被占用，发生冲突，放入 H(H(68)+1)= H(3)= 3 % 11=3。

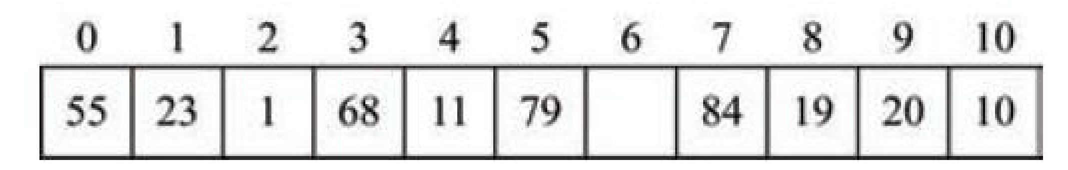
④ 20、84、55：直接放入其对应的散列地址 9、7、0。

⑤ 11：散列地址 0 已被占用，其后的地址 1、2、3 也被占用，经过四次冲突处理，$d_i=1,2,3,4$，最终放入没有占用的 4 号地址。

⑥ 10：直接放入其对应散列地址 10。

⑦ 79：散列地址 2 已被占用，其后的地址 2、3 也被占用经过三次冲突处理，$d_i=1,2,3$，最终放入没有占用的 5 号地址。

结果如图 5.9 所示。



0	1	2	3	4	5	6	7	8	9	10
55	23	1	68	11	79		84	19	20	10

图 5.9　线性探测处理冲突构造的散列表

（2）二次探测再散列：$d_i=1^2,-1^2,2^2,-2^2,\cdots,\pm k^2$ $(k\leqslant n/2)$。

关键字序列{19,23,1,68,20,84,55,11,10,79}，按照散列函数 H(key)=key % 11 和二次探测再散列处理冲突求解散列表。

解：各关键字的散列函数值同线性探测再散列求解的结果，各关键字的散列函数值

和存放位置如下。

① 19、23：直接放入其对应散列地址 8 和 1。

② 1：和 23 为同义词，发生冲突，放入 $H(H(1)+1^2)= H(2)= 2 \% 11=2$ 中。

③ 68：散列地址 2 已被占用，发生冲突，放入 $H(H(68)+1^2)= H(3)= 3 \% 11=3$ 中。

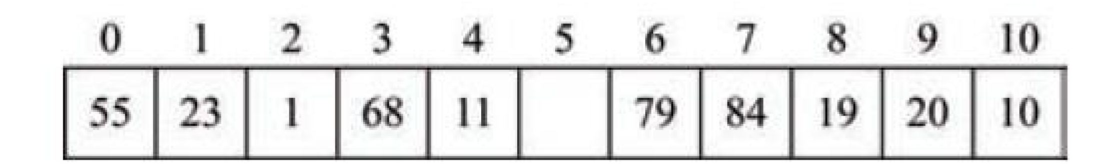
④ 20、84、55：直接放入其对应散列地址 9、7、0 中。

⑤ 11：散列地址 0 已被占用，$0-1^2=-1$ 为非法地址；$0+2^2=4$，该地址没有占用，经过三次冲突处理，$d_i=1^2,-1^2,2^2$，最终放入没有占用的 4 号地址。

⑥ 10：直接放入其对应散列地址 10。

⑦ 79：散列地址 2 已被占用，发生冲突，经过三次冲突处理，$d_i=1^2,-1^2, 2^2$，最终放入没有占用的 6 号地址。

结果如图 5.10 所示。



0	1	2	3	4	5	6	7	8	9	10
55	23	1	68	11		79	84	19	20	10

图 5.10　二次探测处理冲突构造的散列表

（3）随机探测再散列；d_i=伪随机数字序列，较少用。

开放定址法的特点是简单易算，但受表长限制，超过表长的数据元素需要另行处理，而且删除操作比较麻烦。

2．再散列法

$$H(key)=H_i(key) \qquad i=1,2,3,\cdots,k$$

H_i 为不同的散列函数，即在同义词产生地址冲突时使用另一个散列函数地址，直到不冲突为止。该方法大大增加了计算时间。

3．链地址法

散列表的每个记录增加链域，将所有关键字为同义词的记录存储在同一线性链表中，假设某散列函数产生的散列地址在区间[0,m−1]内：

（1）设立一个指针型向量 hashLink[m]，其每个分量的初始状态都是空指针。

（2）散列地址为 i 的记录都插入到头指针为 hashLink[i] 的链表中。

示例：将一组关键字{19,23,1,68,20,84, 55,11,10,79}，按照散列函数 H(key)=key % 11 和链地址法处理冲突求解散列表。

解：各关键字的散列函数值同线性探测再散列求解的结果，图 5.11 为按照要求构造的散列表。

4．公共溢出区

假设散列函数的值域为[0,m−1]，向量 baseTable[0,m−1]为基本表，每个分量存放一个记录，另设向量 overflowTable[0,v−1]为溢出表。所有关键字和基本表中关键字为同义词的记录，均填入溢出表。

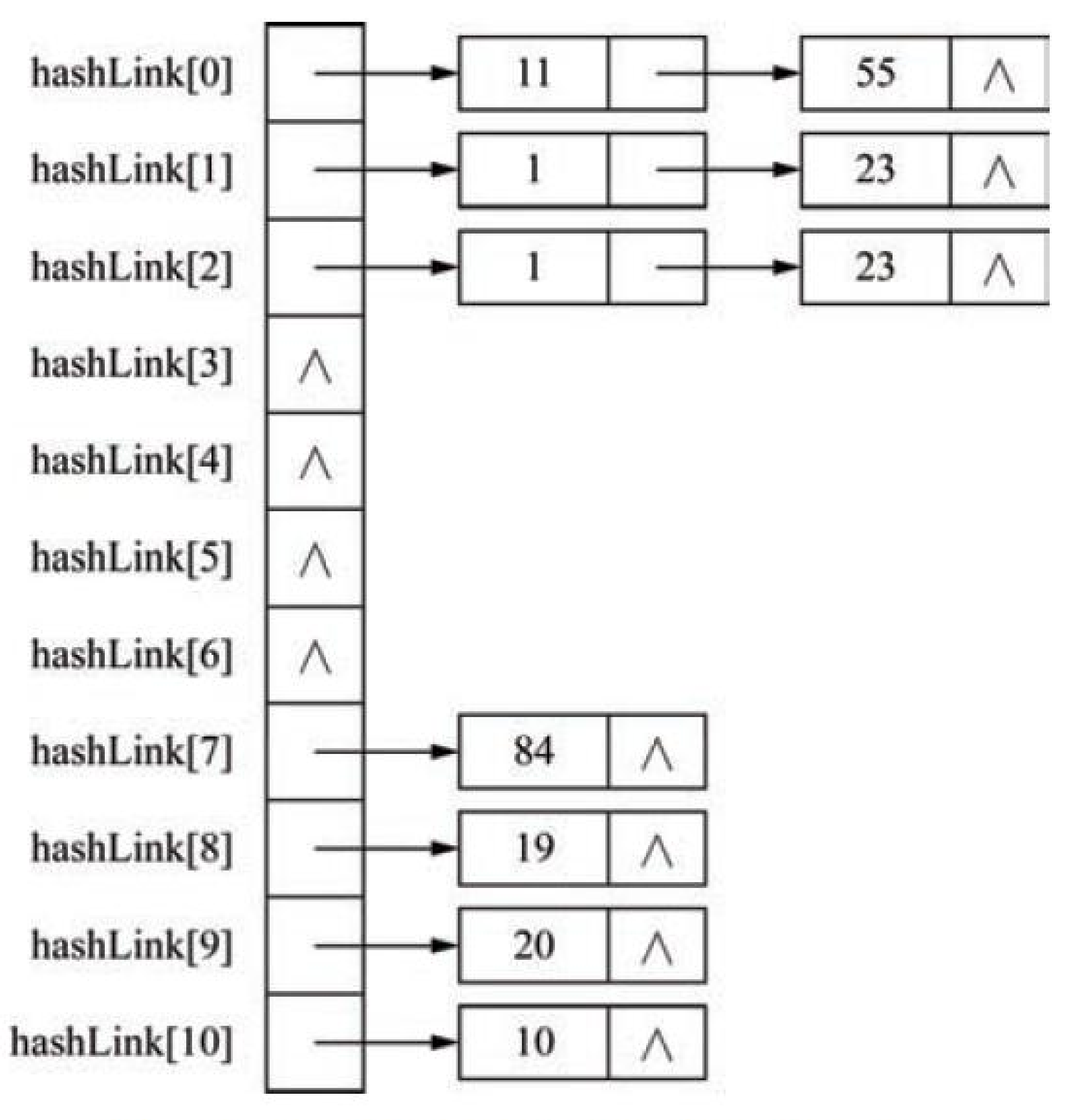


图 5.11　链地址法处理冲突构造的散列表

5.7.4　填充因子

填充因子 α 表示散列表“装满”的程度，定义如下：

$$\alpha = \text{散列表中的记录个数} \,/\, \text{散列表表长}$$

填充因子 α 的作用如下。

（1）α 和几率成正比，其值越大，冲突几率越高。

（2）散列查找的平均查找长度 $ASL_{散列}$ 和 α 有关，与散列表中的记录个数 n 无关。

（3）无论 n 为何值，总能找到一个 α，使 $ASL_{散列}$ 限定在约定的范围。

5.8　本 章 小 结

本章介绍了各种查找算法，重点理解算法的本质、使用场合、对存储结构的要求等，熟练掌握查找过程，掌握查找算法，能够正确求解各查找算法的平均查找长度 ASL，能对算法性能进行分析。

第 6 章　排序

本章学习目标

- 了解排序的相关概念。
- 深入理解各种排序的基本思想。
- 熟练掌握各种排序方法的具体操作过程及伪代码。
- 熟记所有排序算法的时间复杂度和空间复杂度。
- 熟练掌握各种排序的内涵并能明确区别各自特点。

6.1 本章导读



排序

6.1.1 知识结构

本章知识结构如图 6.1 所示，加粗框中的内容需要考生重点理解并掌握。

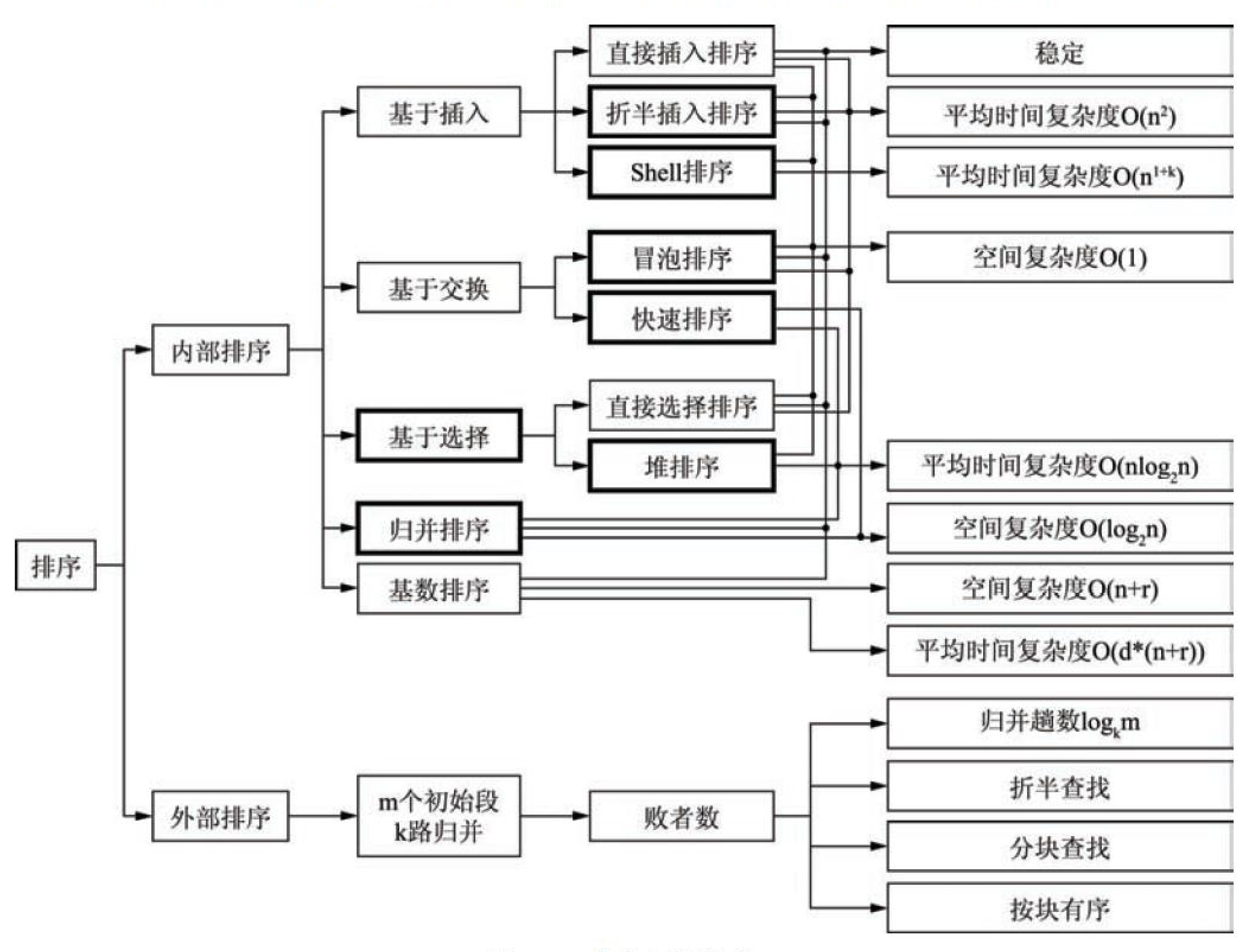


图 6.1　本章知识结构

6.1.2　命题规律

1．命题规律

（1）本章为历年各高校硕士研究生招生考试的重点，命题形式既有客观题又有主观题。

（2）出题形式灵活，既可单独命题，也可联合命题。

（3）内部排序的算法思想及实现（有可能限定时间复杂度或空间复杂度）、排序的稳定性分析、时间复杂度、空间复杂度分析、各排序过程等均为重点考查内容。

2．命题趋势

本章在全国硕士研究生入学考试的重要性地位近年不会改变。例如，与排序相关的基本概念、各种排序的时间复杂度及其稳定性等易出客观题，而各种有关排序过程的客观题、主观题均可能出现。

6.2　基 本 概 念

1．定义

排序又称分类，是计算机程序设计中的一种重要操作，功能为将一组数据元素或记录按某数据项（关键字）的值排列有序的过程。除非特殊说明，本章有序指的是从小到大的升序，且假设待排序初始记录个数为 n。定义如下。

```
#define  MAXSIZE  100
typedef  int  keyType;
typedef  struct{
   keyType  key;                   // 关键字
   infoType  otherinfo;            // 记录的其他信息
}recordType;                       // 记录类型
typedef  struct{
   recordType  r[MAXSIZE+1];       // 存放记录的顺序表，第 0 个位置保留
   int  length;                    // 当前有效记录个数
}sortList;
```

2．排序方法的稳定性

若任意两个关键字相同的记录 R_i、R_j。排序之前 R_i 在 R_j 之前，排序之后 R_i 依然在 R_j 之前，则称该排序方法是稳定的，否则，该排序方法是不稳定的。

3．排序类型

根据排序的数据存放位置可分为 2 类：

（1）内部排序：待排序记录存放在内存中所进行的排序过程。

（2）外部排序：待排序记录数量特别大，内存仅存放部分记录，在排序过程中尚需对外存进行访问的读取其他记录的排序过程。

本章主要讨论内部排序。

根据排序过程的主要操作可分为 4 类：

（1）插入排序：将记录按照关键字插入一个有序序列的排序过程。

（2）交换排序：按照记录关键字的大小，通过交换记录位置完成排序的过程。

（3）选择排序：依次将当前记录序列的最小或最大关键字放置到其最终有序序列位置的排序过程。

（4）归并排序：将 i 个记录看作一个有序子序列（$i=2^k$，$k\in[0,\log_2 n]$），将有序子序列两两合并为长度加倍的更长有序序列，最终 i=n 的排序过程。

6.3 插 入 排 序

6.3.1 直接插入排序

1．基本思想

简单排序方法之一，将一个记录按关键字有序原则插入长度为 1 的有序表，得到长度为 l+1 的有序表的排序方法。具体操作如下。

（1）第 1 个记录为长度为 1 的有序表 sl_1。

（2）将第 2 个记录按约定有序规则插入 sl_1，形成长度为 2 的有序表 sl_2。

（3）依此类推，将第 i 个记录按约定有序规则插入 sl_i，形成长度为 i+1 的有序表 sl_{i+1}。重复 n−1 次，可得到长度为 n 的有序表 sl_n，直接插入排序结束。

2．排序过程示例

如图 6.2 所示为直接插入排序示例。

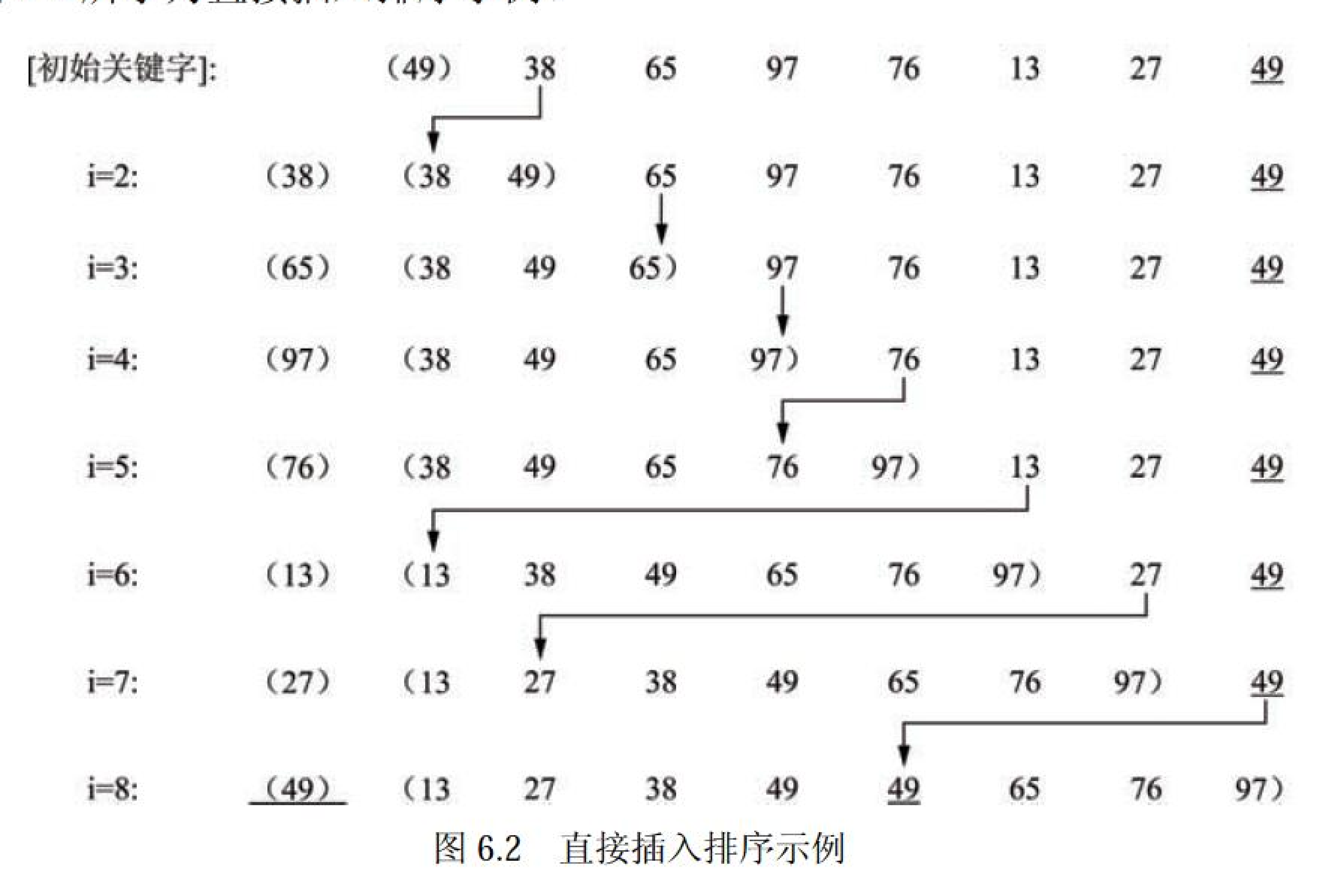


图 6.2 直接插入排序示例

3. 伪代码

```
void directInsertSort(sortList &L) {
    for(i=2;i<=L.length;i++){
        L.r[0]= L.r[i];
        for(j=i-1; L.r[0].key < L.r[j].key; j--)
                                    // 本章以升序为例，降序<改为 > 即可
            L.r[j+1]=L.r[j];
        L.r[j+1]=L.r[0];
    }
}
```

4. 性能分析

（1）空间复杂度：需一个记录的辅助存储空间（L.r[0]），故空间复杂度为 O(1)。

（2）时间复杂度：排序的主要操作为关键字的比较和记录移动。

① 当待排序列中记录按关键字非递减有序（正序）时，所需关键字比较次数为 n−1（最少次数），移动 2(n−1)（最少次数，岗哨和回填的移动次数）。

② 当待排序列中记录按关键字非递增有序（逆序）时，所需关键字比较次数为 (n+2)(n−1)/2（最多次数），记录移动次数(n+4)(n−1)/2（最多次数）。

③ 当待排序记录随机存放时，取上述两种情况的均值，约为 $n^2/4$。

④ 因此，直接插入排序的时间复杂度为 $O(n^2)$。

（3）稳定性：自后向前比较，不会改变等值关键字的记录相对位置，为稳定排序方法。

（4）适用场合

① 顺序存储。

② 链式存储：有无访问的顺序性，链式有序意义不大。

6.3.2 折半插入排序

1. 基本思想

直接插入排序的第 i 趟排序是将 L.r[i]插入到长度为 i−1 的有序表，形成长度为 i 的有序表的过程。由于是有序表，查找 L.r[i]在长度为 i 的有序表的最终位置可以从第 i−1 个记录往前依次比较，也可以利用折半查找方法快速定位，后者称为折半插入排序。显然，折半插入排序比直接插入排序需要的比较次数少。

2. 排序过程

参考 5.3.2 节的折半查找算法确定 L.r[i]在有序表中的位置，然后将其放入该处。

3. 伪代码

```
void halfInsertSort(sortList &L) {
    for(i=2;i<=L.length;i++){
        L.r[0]=L.r[i];
```

```
        low=1;
        high=i-1;
        while(low<=high) {
            mid=(low+high)/2;
            if( L.r[0].key < L.r[mid].key )
                high=mid-1;
            else
                low=mid+1;
        }
        for(j=i-1;j>=low;j--)
            L.r[j+1]=L.r[j];
        L.r[low]=L.r[0];
        }
    }
```

4. 性能分析

（1）空间复杂度：需一个记录的辅助存储空间（L.r[0]），故空间复杂度为 O(1)。

（2）时间复杂度：排序的主要操作为关键字比较和记录移动。

① 关键字比较操作的时间复杂度为 $O(n\log_2 n)$。

② 记录移动操作的时间复杂度为 $O(n^2)$。

由此，折半插入排序的时间复杂度为 $O(n^2)$。

（3）稳定性：折半插入排序是稳定的排序方法。

（4）适用于顺序存储的情况。

6.3.3 希尔排序

1. 基本思想

希尔排序又称“缩小增量”排序、Shell 排序，基本思想为先将整个待排记录按给定步长（增量）分割为若干子序列，并对子序列分别进行直接插入排序。待整个记录“基本有序”时，再对全体记录进行一次直接插入排序。具体过程如下。

（1）选择一个步长（增量）序列 $s=\{s_1, s_2,\cdots, s_k\}$，其中 $s_k=1$。

（2）依次按 s_i，对待排序列进行第 i 趟排序，共 k 趟。第 i 趟排序流程如下：

① 根据步长（增量）s_i，将整个序列分成长度为 n/s_i 的 s_i 个子序列。

② 对每个子序列进行直接插入排序。

③ i=k，即步长为 1 时，整个序列基本有序，对长度为 n 的有序表进行直接插入排序，希尔排序结束。

2. 排序过程

待排序关键字序列为{39,80,76,41,13,29,50,78,30,11,100,7,41,86}（其中两个关键字值均为 41，用是否加下划线区分），步长（增量）序列 s={5,3,1}，则希尔排序过程如图 6.3 所示。

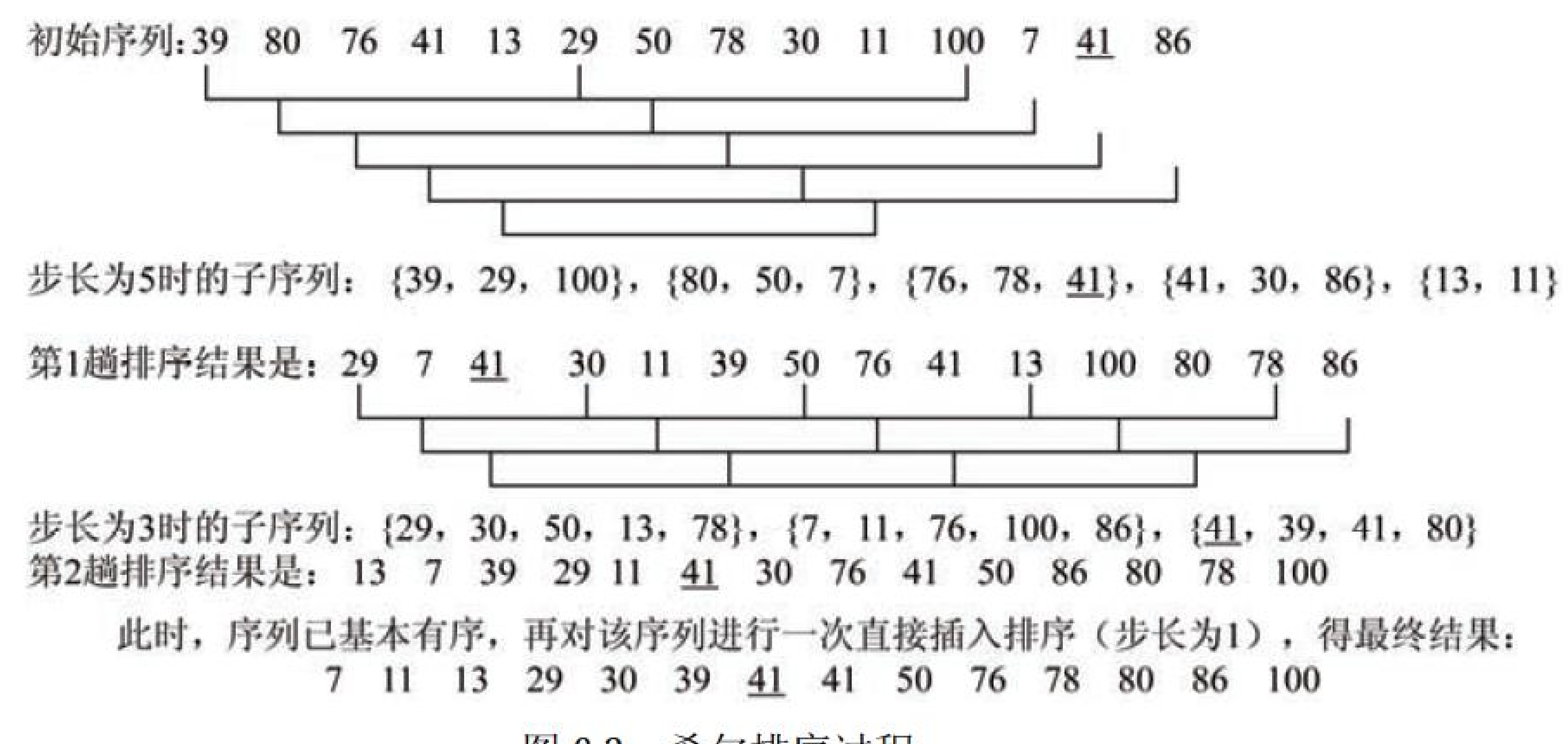


图 6.3　希尔排序过程

3. 伪代码

```
void shellInsert (sortList &L, int t) { // 按步长（增量）t 进行希尔排序
    for(i=t+1;i<=L.length;i++)
        if(LT(L.r[i].key,L.r[i-t].key)) {
            L.r[0]=L.r[i];
            for( j=i-t; j>0 && L.r[0].key < L.r[j].key; j-=t )
                L.r[j+t]=L.r[j];
            L.r[j+t]=L.r[0];
        }
}
void shellInsertSort(SqList &L,int s[], int s[], int k) {
                                        // 按步长（增量）序列 s[0,1,…,k-1]
                                        // 对顺序表 L 进行希尔排序
    for( i=0; i<k; i++)
        shellInsert(L,s[i]);
}
```

4. 步长（增量）序列

步长（增量）序列和希尔排序的时间复杂度有关，所以其选择至关重要。但目前对步长（增量）序列的选取没有明确、规范的方法。一般步长因子必须遵守以下原则：

（1）步长序列中的所有步长除了 1 之外没有公因子。

（2）最后一个步长因子必须为 1。

5. 性能分析

（1）空间复杂度：需一个记录的辅助存储空间（L.r[0]），故空间复杂度为 O(1)。

（2）时间复杂度：和步长（增量）序列有关，目前没有准确答案。优于直接插入排序，最坏情况为 $O(n^2)$。

（3）稳定性：比较跨度较大，为不稳定排序方法。

（4）适用于顺序存储的情况。

6.4 交换排序

交换排序包括冒泡排序和快速排序两种方法。

6.4.1 冒泡排序

1．基本思想

冒泡排序是简单排序方法之一，很多编程语言将其作为循环章节的示例。将一个待排序的序列，通过两两相邻进行比较，将关键字值小的记录放在前面，大的放在后面的排序方法。具体如下。

（1）待排序序列长度 l 保存于 length，即 length=l。

（2）通过两两相邻比较交换，将长度为 length 的待排序序列 L.r[1]–L.r[length]中的最大值放入 L.r[length]。

（3）待排序序列长度减 1，即 length=length−1。

（4）重复（2）、（3），直到某趟比较没有发生交换或 length=2，冒泡排序结束。

2．排序过程

图 6.4 为冒泡排序示例。

初始关键字	第1趟排序后	第2趟排序后	第3趟排序后	第4趟排序后	第5趟排序后	第6趟排序后
49	38	38	38	38	13	13
38	49	49	49	13	27	27
65	65	65	13	27	38	38
97	76	13	27	49	49	
76	13	27	49	49		
13	27	49	65			
27	49	76				
49	97					

图 6.4　冒泡排序示例

3．伪代码

```
void bubbleSort(sortList &L) {
    changed=1;
    for( i=1; i<L.length && changed; i++){
       changed=0;
       for( j=1; j<=L.length-i; j++)
           if( L.r[j] > L.r [j+1] ) {
             L.r[0]= L.r[j];
             L.r[j]= L.r[j+1];
             L.r[j+1]= L.r[0];
             changed=1;
           }
     }
}
```

4．性能分析

（1）空间复杂度：需一个记录的辅助存储空间（L.r[0]），故空间复杂度为 O(1)。

（2）时间复杂度：排序的主要操作为关键字比较和记录交换。

① 当待排序列中记录按关键字非递减有序（正序）时，所需比较次数为 n−1（最少次数），无须交换（最少次数）。

② 当待排序列中记录按关键字非递增有序（逆序）时，所需关键字比较次数为 n(n−1)/2（最多次数），记录交换次数 n(n−1)/2（最多次数）。

③ 当待排序记录随机存放时，取上述两种情况的均值，约为 n(n−1)/4。

由此，冒泡排序的时间复杂度为 $O(n^2)$。

（1）稳定性：自前向后两两相邻进行比较，不会改变等值关键字的记录相对位置，为稳定排序方法。

（2）适用于顺序存储的情况。

6.4.2　快速排序

1．基本思想

（1）通过一趟排序将长度为 l 的待排序列[L.r[1],L.r[l]]分为 3 个部分：

① L.r[i]：有序表的第 i 个记录，划分结点。

② 子序列 [L.r[1],L.r[i−1]]：L.r[k].key≤L.r[i] .key，k∈[1, i−1]。

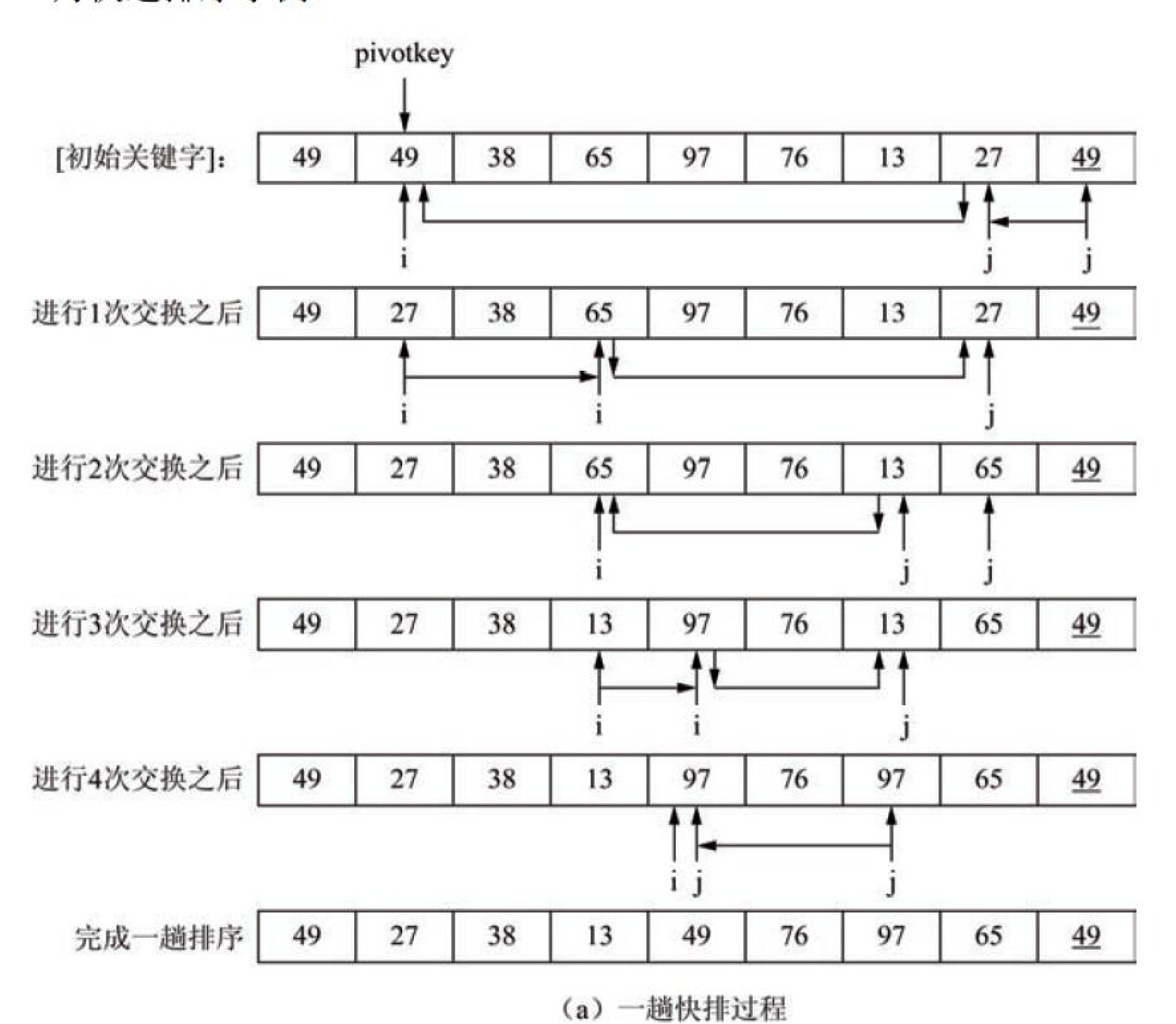
③ 子序列 [L.r[i+1],L.r[l]]：L.r[k].key≥L.r[i] .key，k∈[i+1, l]。

（2）对长度为 i−1 的序列 [L.r[1],L.r[i−1]]，以及对长度为 l−i 的序列 [L.r[i+1],L.r[l]]分别进行类似（1）的排序。

（3）重复（1）、（2），直到各个子序列为空或仅含一个记录为止。

2．排序过程

图 6.5 为快速排序示例。



（a）一趟快排过程

图 6.5　快速排序示例

初始状态	{49	38	65	97	76	97	65	49}
一次划分后	{27	38	13}	49	{76	97	65	49}
分别进行快排	{13}	27	{38}					
	结束		结束		{49	65}	76	{97}
								结束
					49	{65}		
						结束		
有序序列	{13	27	38	49	49	65	76	97}

（b）排序全过程

图 6.5（续）

3. 伪代码

```
unsigned partition(sortList &L, unsigned low, unsigned high) {
                              // 以 low 位置上的元素为标准进行一次划分
    L.r[0]=L.r[low];
    midkey=L.r[low].key;
   while(low<high) {
      while(low<high && L.r[high].key >= midkey)
         high--;
      L.r[low]=L.r[high];
      while(low<high && L.r[low].key <= midkey)
         low++;
      L.r[high]=L.r[low];
   }
   L.r[low] = L.r[0];
   return low;
}
void quickSortProcedure(sortList &L, unsigned low, unsigned high) {
   if(low<high){
      mid = partition(L,low,high);
      quickSortProcedure (L,low,mid-1);
      quickSorProceduret(L,mid+1,high);
   }
}
void quickSort(sortList &L) {
   quickSortProcedure (L,1,L.length);
}
```

4. 性能分析

（1）空间复杂度：算法的递归性需要借助栈实现。

① 最好空间复杂度为 $O(\log_2 n)$。

② 最坏空间复杂度为 $O(n)$。

③ 平均空间复杂度为 $O(\log_2 n)$。

（2）时间复杂度。

① 最坏时间复杂度为 $O(n^2)$。

② 平均时间复杂度为 $O(n\log_2 n)$。

（3）稳定性：不稳定。

（4）适用于顺序存储。

5．说明

（1）通常快速排序被认为是所有同数量级（$O(n\log_2 n)$）排序方法中平均性能最好的方法。

（2）若待排序列基本有序，且首元素作为划分结点，则快速排序蜕变为冒泡排序。

（3）一般可选取首元素、尾元素及中间元素三者居中者作为划分结点。

（4）快速排序的平均性能不可能达到 O(n)。

6.5　选择排序

选择排序分为直接选择排序和堆选择排序两种。

6.5.1　直接选择排序

1．基本思想

将长度为 len 的待排序序列中具有最小或最大（升序或降序）关键字值的记录的第 i 个记录交换（初值为 1），len 减 1，i 加 1，直至序列按关键字有序，具体如下（以升序为例）。

（1）i=1。

（2）将长度为 len−i+1 的待排序序列中具有最小关键字值的记录的第 i 个记录交换。

（3）i= i+1。

（4）重复（2）、（3），直至 i= len−1。

2．排序过程

考生自行通过直接选择排序将图 6.5 的初始无序序列排列有序。

3．伪代码

```
void  selectSort(sortList  &L) {
     for(i=1; i<L.length; i++){
         min=i;
         for(j=i+1; j<L.length; j++)
             if( L.r[min].key > L.r[j].key )
                 min=j;
         if(min!=i){
             L.r[0]= L.r[i];
             L.r[i]= L.r[min];
             L.r[min]= L.r[0];
         }
     }
}
```

4．性能分析

（1）空间复杂度：需一个记录的辅助存储空间（L.r[0]），故空间复杂度为 O(1)。

（2）时间复杂度为 $O(n^2)$。

（3）稳定性：稳定的排序方法。

（4）适用于顺序存储的情况。

6.5.2 堆选择排序

1．基本概念

（1）定义：n 个元素的序列$\{k_1,k_2,\cdots,k_n\}$，当且仅当满足下列条件时，称为堆。

$$\begin{cases}k_i \leqslant k_{2i}\\ k_i \leqslant k_{2i+1}\end{cases} \text{或} \begin{cases}k_i \geqslant k_{2i}\\ k_i \geqslant k_{2i+1}\end{cases} \left(i=1,2,\cdots,\left[\frac{n}{2}\right]\right)$$

（2）特点：

① 一般堆的存储结构为完全二叉树顺序存储形式。

② 完全二叉树中所有非终端结点 k_i 的关键字值均不大于（或不小于）其左孩子结点 k_{2i}、右孩子结点 k_{2i+1} 的关键字的值（如果 k_{2i}、k_{2i+1} 存在）。

③ 若根结点的关键字值为整个序列关键字的最大值，称为大根堆；如果根结点的关键字值为整个序列关键字的最小值，称为小根堆或小顶堆。

（3）建堆即调整完全二叉树，使其符合堆定义的过程。

（4）建堆过程即堆排序过程。

（5）示例。

图 6.6 为大根堆和小根堆及对应二叉树形式。

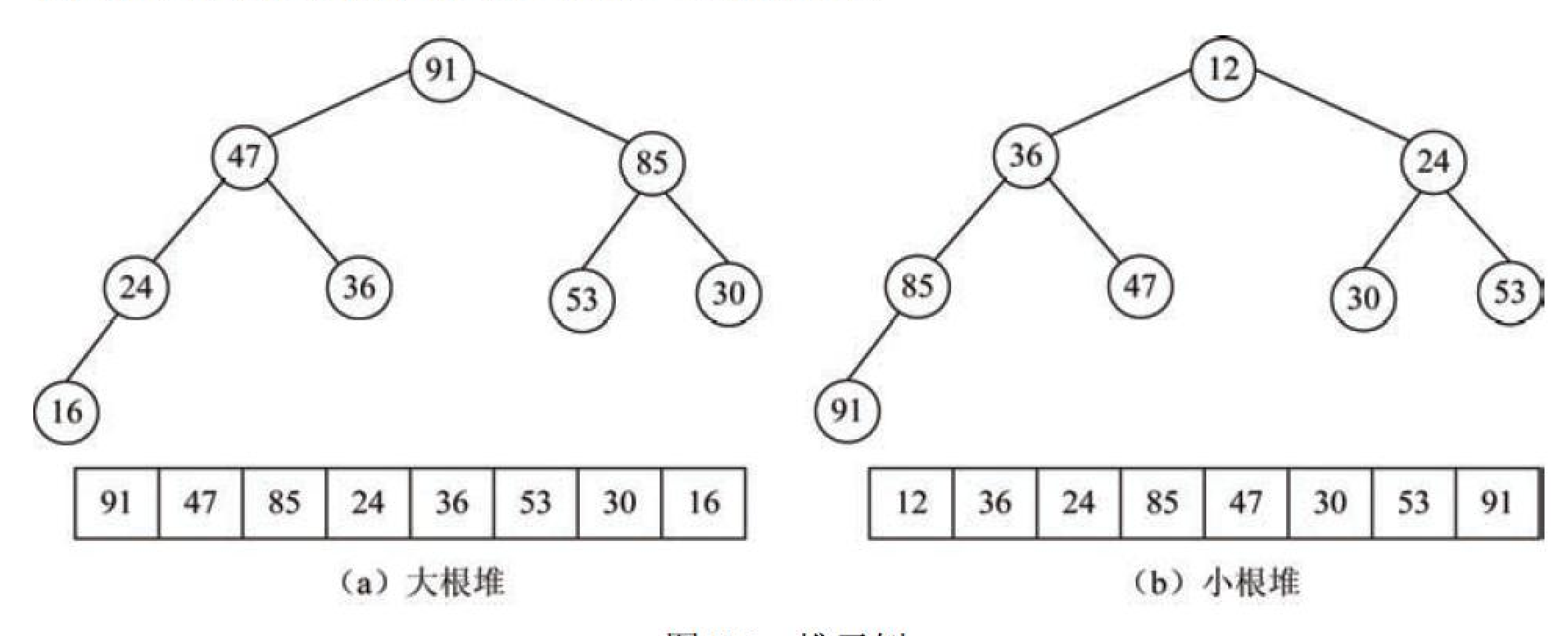


图 6.6　堆示例

2．基本思想

（1）对含 n=l 个记录的序列按关键字建堆。

（2）输出堆顶记录（关键字最小或最大（小根堆或大根堆）的记录）。

（3）剩余 n−1 个元素调整为堆。

（4）重复（2）、（3），直到所有记录输出。

这个过程称为堆排序。实现堆排序必须解决以下两个问题。

（1）将 n 个记录的序列按关键字建堆。

（2）输出堆顶记录，调整剩余 n-1 个记录，使其按关键字成为一个新堆（以大堆根为例）。

① 将堆顶记录 r.L[1]放置到最后记录 r.L[k] 的存储位置，新的最后记录（关键字值最大）脱离堆，并输出。

② 保存最后记录 r.L[k]。

③ 选择堆顶记录 r.L[1]左、右子树根结点关键字值较大值的，移动到堆顶位置，

④ 对被移动的结点进行③的操作。

⑤ 依次重复③，④，直到某个结点的左、右子树为空。

⑥ 将②保存的记录放置到⑤找到的左、右子树为空的结点位置，堆调整结束。

3．排序过程

图 6.7 为大根堆的一次调整过程示例。

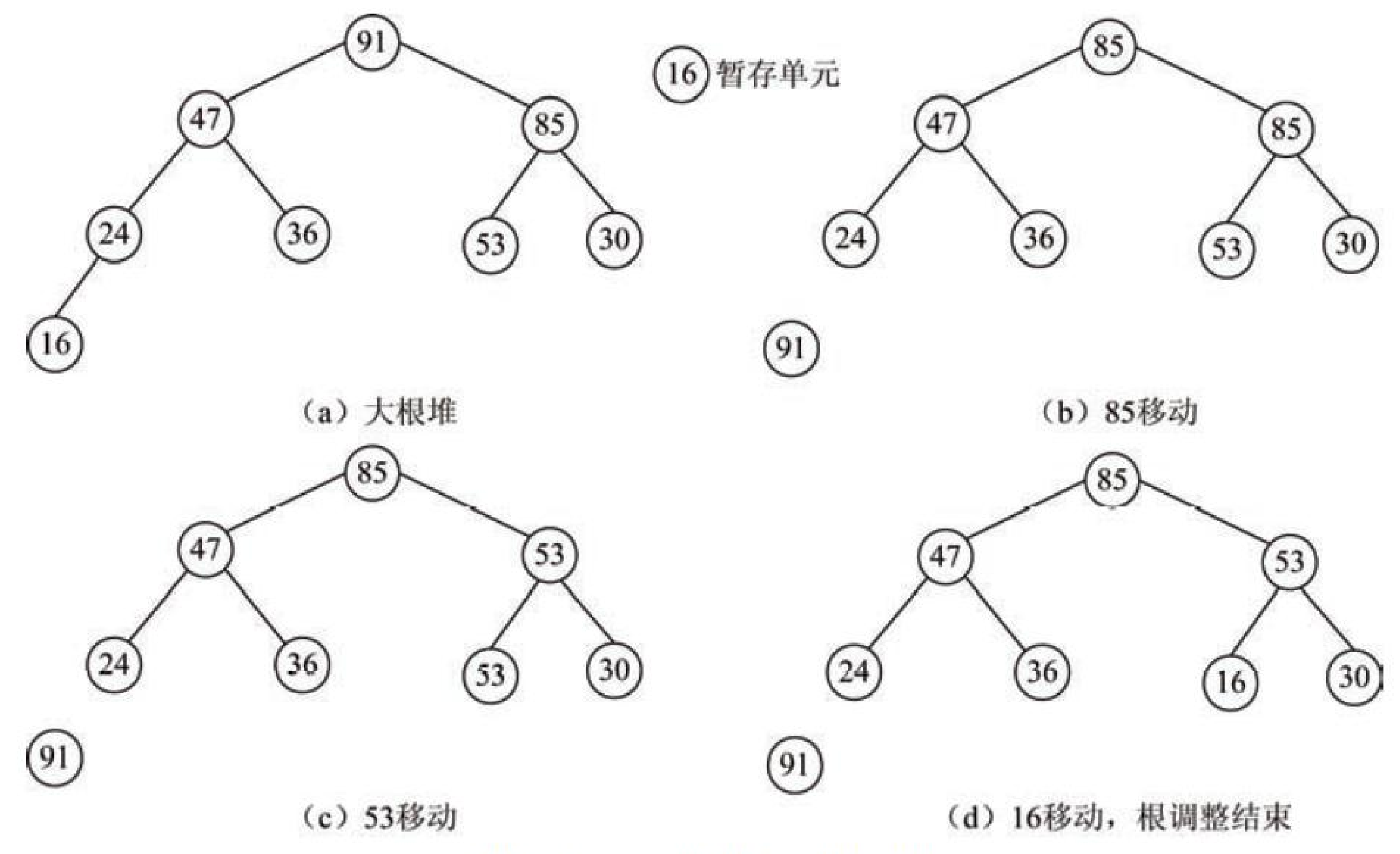


图 6.7　大根堆调整过程示例

图 6.7 的说明如下。

（1）图 6.7（a）为初始大堆。

（2）图 6.7（b）为：

① 堆顶 96 所在记录和堆最后记录 16 所在的结点交换位置，并脱离堆。

② 新堆顶 16 所在记录保存至暂存单元。

③ 堆顶左、右子树根结点关键字 47、85 中大者 85 所在记录放入堆顶。

（3）图 6.7（c）为原 85 所在结点左、右子树根结点关键字 53、30 中大者 53 所在记录放入原 85 所在结点。

（4）图 6.7（d）为暂存单元的记录放入原 53 所在结点。

（5）对图 6.7（d）的堆重复类似（1）～（4）的操作可找到次大关键字，并将剩余结点建堆。

（6）依此类推，可通过堆排序按关键字从大到小输出有序序列。

4．伪代码

```
void  heapAdjust(sortList  &H, int start, int end) {
                                        // 调整以 start 结点为根的子树为堆
    H.r[0]= H.r[start];                 // start 结点存放记录暂存于序列 0 号单元
    t=start;
    for(i=2*t; i<=end; i*=2) {          // t 为 i 的双亲结点
        if( i<end && H.r[i].key<H.r[i+1].key )
            i++;                        // H.r[i].key 为 t 左、右子树根结点关键字值的
                                        // 较大者
        if( H.r[0].key>=H.r[i].key )
            break;
          H.r[t]=H.r[i];                // 较大子树根结点复制至双亲结点
        t=i;                            // 保存较大子树根结点序号
    }
    H.r[t]= H.r[0];                     // 原堆顶记录放入最终位置
}
void  heapSort(sortList  &H) {
     for(i=H.length/2; i>0; i--)
        heapAdjust(H,i,H.length); // 创建初始堆
    for(i=H.length;i>1;i--){
        H.r[0]=H.r[i];                  // 交换堆顶元素（第 1 个）和最后一个堆元素（第 i
                                        // 个）的内容
        H.r[i]=H.r[1];                  // 原堆顶元素（现最后一个堆元素）在下
        H.r[1]=H.r[0];                  // 一轮循环退出调整建堆，即出堆
        heapAdjust(H,1,i-1);            // 其余 i-1 个元素调整建堆
    }
}
```

5．性能分析

（1）空间复杂度为 O(1)。

（2）时间复杂度为 $O(n\log_2 n)$。

（3）稳定性为不稳定排序方法。

（4）适用于顺序存储、有较多记录文件的情况，不适合小文件。

6.6 归并排序

1．基本思想

归并排序为将两个或两个以上的有序表合并为一个包含更多记录的有序表，二路归并排序流程如下。

（1）n=l，length=1，即将含 l 个记录的待排序列长度 l 暂存于 n，初始有序子序列长度 1 暂存于 length。

（2）将长度为 n 的待排序列看作 n 个有序的、长度为 length 的子序列。

（3）n 个有序的、长度为 length 子序列两两合并，得到 n/2 个长度为 length 或 2*length 的有序子序列。

（4）n=n/2，length=2*length，重复（2）、（3），直到整个待排序列为有序序列为止。

2．排序过程

图 6.8 是一个二路归并排序的例子。

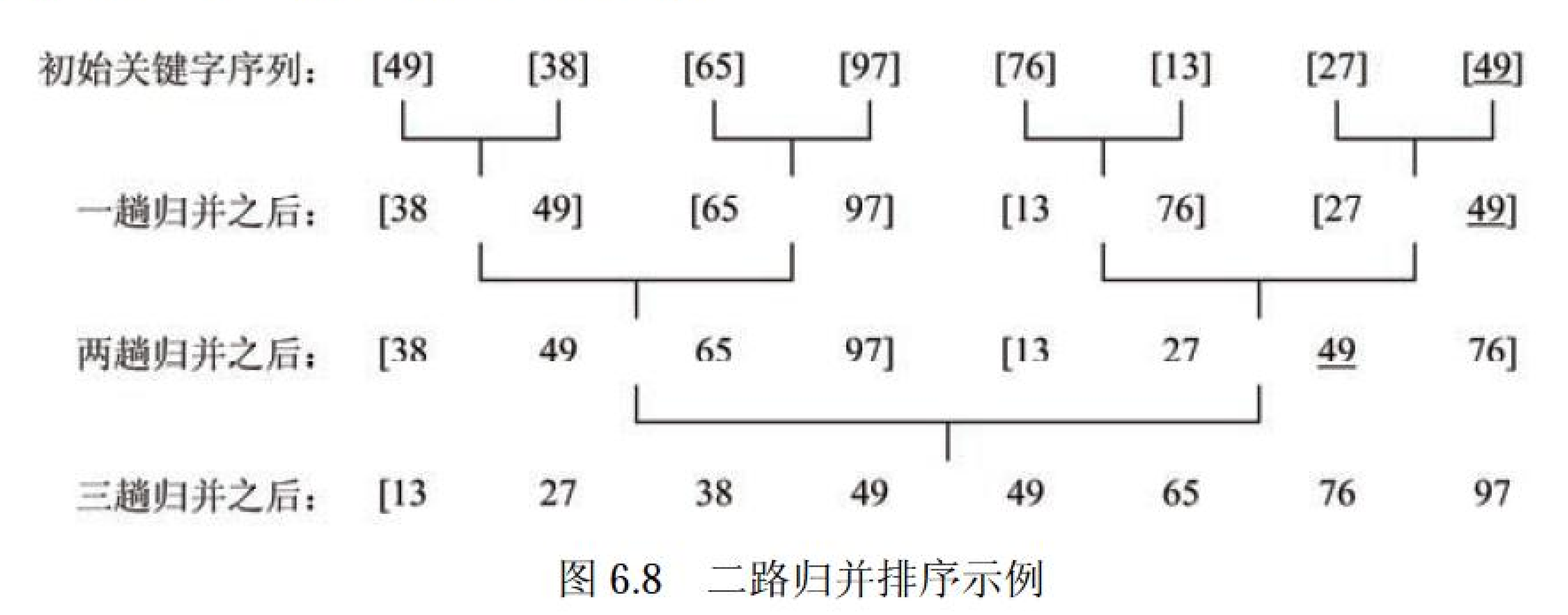


图 6.8　二路归并排序示例

3．伪代码

```
void merge(sortList SR[], int i, int m, int n) {
                                        // 将有序序列[SR[i]、SR[m]]和[SR[m+1],
                                        // SR[n]]合并为有序序列[SR[i], SR[n]]
   for(s=1; s<= L.length; s++)
      tL.r[s]= SR.r[s];                 // tL 为辅助数组
    for(k=i, j=m+1;  i<=m && j<=n;  k++){
        if(tL[i].r.key< tL[j].r.key)
          SR[k]= tL[i++];
        else
          SR[k]= tL[j++];
   }
   if(i<=m)
      for(j=i, t=k; j<=m; j++,t++)
         SR[t]=tL[j];
   else                                 // j<=n
      for(i=j, t=k; i<=n; i++,t++)
         SR[t]=tL[i];
}
void mSortProcedure (sortList SR[], int start, int end) {
    if(start<end) {
      mid=( start +end)/2;
      mSortProcedure (SR, start, mid);
      mSortProcedure (SR, mid+1,end);
      merge(SR, start, mid, end);
   }
}
void mergeSort(sortList &L) {
      mSortProcedure(L, 1, L.length );
}
```

4. 性能分析

（1）空间复杂度：需和原序列同样大小的辅助存储空间，故空间复杂度为 O(n)。

（2）时间复杂度为 $O(n\log_2 n)$。

（3）稳定性为稳定排序方法。

（4）适用于顺序存储的情况。

6.7 基 数 排 序

和前述主要通过关键字比较和移动记录两种操作完成排序不同，基数排序是一种借助多关键字排序思想对单关键字进行排序的方法，比如扑克牌按花色和面值的排序。

1. 基本思想

基础为多关键字排序，假设有 n 个记录序列$\{R_1, R_2, \cdots, R_n\}$，且每个记录 R_i 中含 d 个关键字（$key_{i1},key_{i2},\cdots,key_{id}$），序列$\{R_1,R_2,\cdots,R_n\}$对关键字列表（$key_{i1},key_{i2},\cdots,key_{id}$）有序指：

任意两个记录 R_i 和 $R_j(1\leqslant i< j\leqslant n)$满足下列关系，

$$(key_{i1},key_{i2},\cdots,key_{id}) < (key_{j1},key_{j2},\cdots,key_{jd})$$

其中 key_{i1} 称为主关键字，key_{i2} 称为第 2 关键字，…，key_{id} 称为第 d 关键字。

通常有两种方法实现多关键字排序。

1）最高位优先（MSD）法

（1）按 key_{i1} 相等原则，将整个序列划分为若干子序列，每个子序列的所有记录具有相同的 key_{i1} 值。

（2）按 key_{i2} 相等原则，将每个子序列划分为若干更短的子序列。

（3）依此类推，直到所有子序列均有序。

（4）将所有有序子序列依次相接，构成一个完整的有序序列。

2）最低位优先（LSD）法

（1）按 key_{id} 相等原则，将整个序列划分为若干子序列，每个子序列的所有记录具有相同的 key_{id} 值。

（2）按 key_{id-1} 相等原则，将每个子序列划分为若干更短的子序列。

（3）依此类推，直到所有子序列均有序。

（4）将所有有序子序列依次相接，构成一个完整的有序序列。

两种方法的具体排序可通过前面的排序方法实现。并且为减少排序所需辅助存储空间，通常采用静态链表作为存储结构，即链式基数排序。

2. 排序过程示例（以 LSD 为例）

（1）分配：从最低位关键字 key_{id} 开始，按关键字值的不同将待排序列的所有记录“分配”到 r 个队列。

（2）收集：各队列按关键字大小顺序连接。

（3）关键字依次向前，（1）、（2）重复 d 次，基数排序结束。

图 6.9 为基数排序示例，关键字列表为百位数、十位数、个位数。

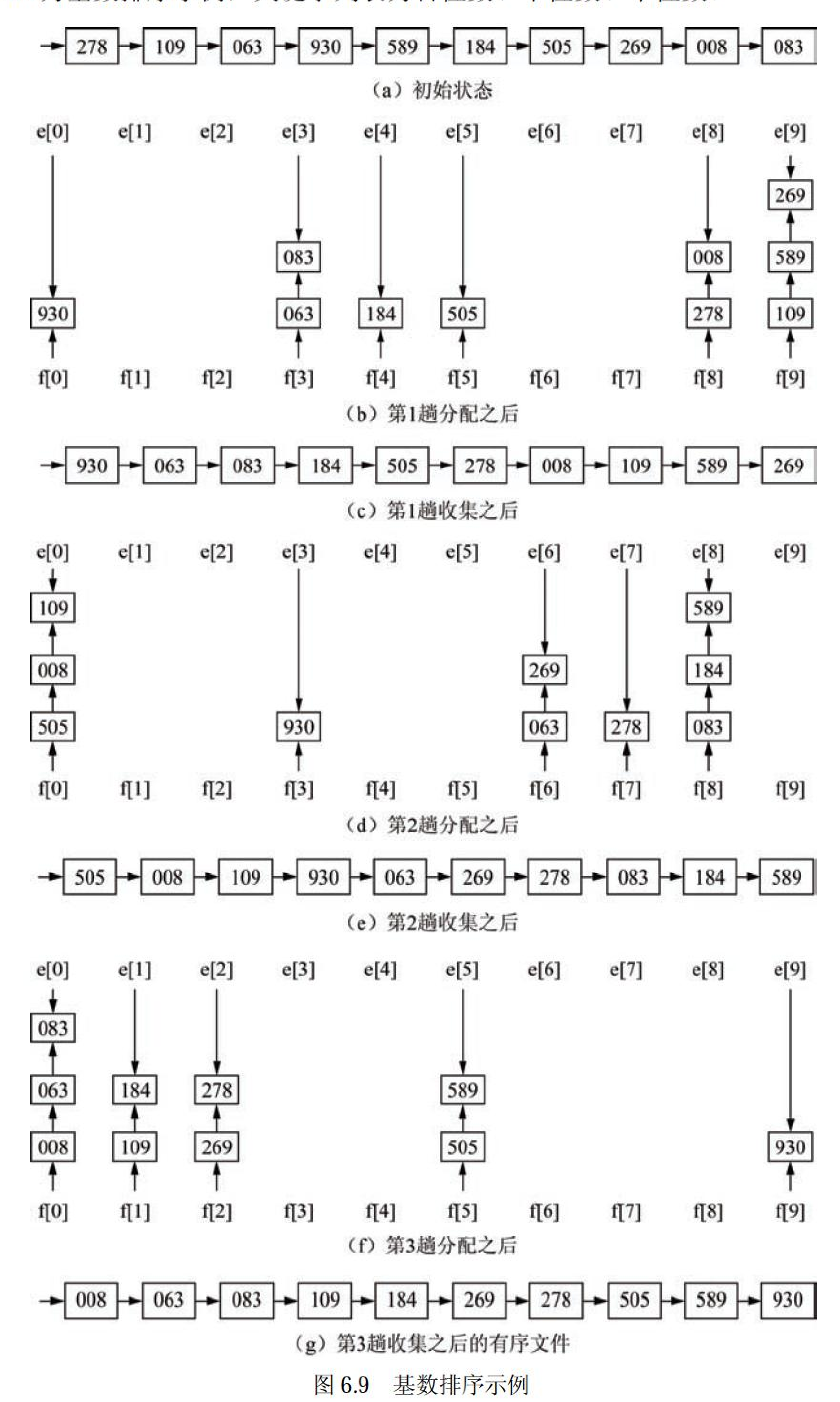


图 6.9　基数排序示例

3. 伪代码

```
#define  MAX_NUM_OF_KEY  6            // 关键字个数最大值
#define  RADIX            10          // 关键字的基数
#define  MAX_SPACE    10000
typedef  struct{
     keyType  key[MAX_NUM_OF_KEY];    //关键字列表
     InfoType  otheritems;
     int       next;
}radixSortNode;
typedef  struct{
    radixSort   r[MAX_SPACE];
    int        keyNum;
    int        recordNum;
}radixSortList;
typedef  int  arrayType[RADIX+1];     //0下标的元素作为中间变量，不存放记录信息
void distributeNode(radixSortList &r, int i, arrayType &f, arrayType &e) {
//f[j]指示第j个队列的第一个记录,e[j]指示第j个队列的最后一个记录,j∈[0,RADIX-1]
    for( j=0; j<=RADIX; j++)
        f[j]=0, e[j]=0;               // 队列为空
     for(p=r[0].next;  p;  p=r[p].next) {
                                      // 将记录序列r中的各记录，分配到当前关键
                                      // 字对应的队列
        j=order(r[p].keys[i]);        // 当前记录（序号为p）的第i个关键字暂存
                                      // 于j
        if(!f[j])                     // 如果第j个队列的第1个记录为空
            f[j] = p;                 // 修改第j个队列的第1个记录的序号为p
        else
            r[e[j]].next = p;         // 如果第j个队列的第1个记录不为空，序号
                                      // p插入该队列尾部
        e[j]=p;                       // 修改第j个队列的最后一个记录序号为p
    }
}
void  collectQueue(radixSortList  &r, int i, arrayType  f, arrayType  e) {
     for(j=0; !f[j]; j++)
        ;                             // 查找第1个非空队列
     r[0].next = f[j];                // 第1个非空队列链接入链表
     t=e[j];
     while(j<=RADIX) {
        for( j=j+1; j<RADIX && !f[j]; j= j+1);
        if(f[j]) {
            r[t].next=f[j];
            t=e[j];
        }
    }
    r[t].next=0;
}
void  radixSort(radixSortList  &L) {
     for(i=0;i<L.keyNum;i++)
        L.r[i].next=0;
     L.r[L.recNum].next=0;
     for(i=0;i<L.keyNum;i++){
        distributeNode(L.r,i,f,e);
        collectQueue (L.r,i,f,e);
    }
}
```

4. 性能分析

设每个记录含 d 个关键字，r 个队列，对 n 个记录进行链式基数排序。

（1）空间复杂度。

① n 个记录的 next 域需要 n 个辅助空间。

② 每个队列需要 f 和 e 两个辅助空间，r 个队列共 2*r 个辅助空间。

故空间复杂度为 O(n+r)。

（2）时间复杂度。

① 一趟收集的时间复杂度为 O(r)。

② 一趟分配的时间复杂度为 O(n)。

③ 分配、收集共 d 趟。

由此，基数排序的时间复杂度为 O(d* (n+r))。

（3）稳定性：稳定排序方法。

（4）适用于静态链表。

6.8 内部排序方法比较

内部排序应用较多，但每种排序方法的难易程度、时间复杂度、空间复杂度、实现难易程度、是否稳定等一般不可兼得。如有的算法稳定，容易理解，实现简单，空间复杂度低，但时间复杂度较高。所以需要熟知各种排序算法特征，方便为实际应用选择合适的排序方法。表 6.1 为各种排序方法的比较，考生需要理解并牢记。

表 6.1　各种内部排序方法比较

排序方法		时间复杂度		空间复杂度	稳定性
		平均	最坏		
简单排序	直接插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
	折半插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
	冒泡排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
	直接选择排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
高级排序	快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定
	希尔排序	$O(n^{1+k})$ $(0<k<1)$	$O(n^{1+k})$ $(0<k<1)$	O(1)	不稳定
	归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	稳定
	基数排序	O(d*(n+r))	O(d*(n+r))	O(n+r)	稳定

对表 6.1 所列内部排序方法从稳定性、空间复杂度及平均时间复杂度三方面进行分析，有如下几点。

1. 稳定性

（1）快速排序、堆排序和希尔排序是不稳定的排序方法。

（2）证明排序方法不稳定，仅需举出一个实例说明即可。

（3）对多关键字记录序列进行基数排序时，最终排序采用稳定排序方法。

2．空间复杂度

（1）归并排序空间复杂度最高，为 $O(n)$。

（2）快速排序为递归过程，空间复杂度为 $O(\log_2 n)$。

（3）基数排序空间复杂度为 $O(n+r)$。

（4）其他排序算法空间复杂度为 $O(1)$。

3．平均时间复杂度

（1）直接插入排序、折半插入排序、冒泡排序和简单选择排序是不稳定的排序方法，平均时间复杂度为 $O(n^2)$。而且直接插入排序和冒泡排序在记录序列按关键字“基本有序”时，实际排序速度很快。

（2）快速排序、堆排序和归并排序平均时间复杂度为 $O(n\log_2 n)$。

（3）基数排序平均时间复杂度为 $O(d*(n+r))$。

（4）希尔排序平均时间复杂度为 $O(n^{1+k})$ $(0<k<1)$。

6.9 外部排序

当待排序记录规模过大、内存空间无法容纳时，需要采用外部排序对存放于外存的大文件进行排序，通过在内外存之间多次交换数据完成整个文件的排序。此时内存主要作为工作空间辅助外存数据的排序，外存作为大文件的存放地。

1．基本思想

由于归并排序不需要将全部记录读入内存即可完成排序，可以解决由于内存空间不足导致无法对大规模记录进行排序的问题，所以外部排序常用的算法是多路归并排序。通过将大文件的数据分为多个内存可以容纳的部分；一次调一部分数据进入内存，通过内部排序方法完成排序；多次重复，将各部分排好序后，再对已排序的各部分进行多路归并完成整个文件的排序，具体算法流程如下。

（1）将待排序序列划分为 m 个长度可不等、规模较小、内存可容纳的子序列，并分别对各子序列用内部排序方法进行排序。

（2）将 m 个有序子序列分别写入 m 个子文件。

（3）对 m 个有序子文件多次采用多路归并方法进行排序，直到所有记录有序。

2．重要概念：败者树

败者树为多路归并排序算法中选最值的方法，可以提高选最值的效率，特点如下（以升序为例）。

（1）败者树是完全二叉树。

（2）一个叶结点对应一个子序列，存放各归并段当前要参加归并的记录。

（3）非终端结点表示其左、右孩子中的“败者”（逆序的记录），胜者去参加更高一级的比赛。

（4）根结点为所有子序列的败者。

（5）根结点的父结点存放最终胜利者，为本次归并排序选出的最小值，输出相关记录信息。

（6）选出元素所在序列的下一个元素替补该元素，进入败者树结点。

（7）沿新进败者树结点往根结点的路径，修正败者树，可选出本次归并的次小者，输出相关记录信息。

（8）重复进行，直到输出各归并段的所有记录。

3．性能评价

（1）时间复杂度：外部排序的时间复杂度涉及很多方面，且分析较为复杂，注意以下几点。

① m 个初始归并段进行 k 路归并，归并的趟数为 $\log_k m$。

② k 路归并的败者树的高度为 $1+\log_2 k$，因此利用败者树从 k 个记录中选最值需要的比较次数为 $\log_2 k$，即时间复杂度为 $O(\log_2 k)$。

③ k 路归并败者树的建树时间复杂度为 $O(k\log_2 k)$。

（2）空间复杂度：算法所有步骤中的空间复杂度均为常量，因此空间复杂度为 O(1)。

6.10　本 章 小 结

本章算法较多，但部分内容及思想前面章节已经介绍。复习过程中注意分类学习，领会思想、熟知过程、牢记时间复杂度和空间复杂度。重点掌握各种排序的基本思想，执行过程，算法设计及不同排序方法之间的比较。能熟练根据给定的数据序列，使用不同的排序方法，写出要求的排序方法的排序过程。难点为快速排序、希尔排序、堆排序、归并排序等。

不同排序方法各有优缺点，没有一种排序方法是完美无缺的，可根据应用条件的特点选择合适的方法，甚至可将多种方法结合使用。以下为部分建议（设待排序记录个数为 n，又称问题规模）。

（1）n 不大的情况适合选用三种简单排序方法（直接插入排序、简单选择排序、冒泡排序）。虽然时间复杂度达 $O(n^2)$，但方法简单易掌握，而且直接插入排序和冒泡排序在记录序列按关键字“基本有序”时，实际排序速度很快。

（2）n 较大，不强求稳定性的情况，可考虑选用快速排序或堆排序。但快速排序在原序列基本有序时，速度反而减慢，时间复杂度达 $O(n^2)$，堆排序时间复杂度稳定。

（3）n 很大，要求排序稳定，且存储容量不受限的情况，适合采用归并排序。

（4）n 值很大且关键字位数较小的情况，采用静态链表基数排序较好。

主要算法总结

算法名称	时间复杂度	空间复杂度
稀疏矩阵快速转置算法	O(nu+tu)　nu:矩阵列数，tu:矩阵非 0 元素个数	O(nu)
一般模式匹配算法	O(n*m) n:主串长度，m:模式串长度	O(1)
KMP 算法	O(n+m)	O(m)
二叉树遍历	O(n)	O(n)
图遍历	O(n+e)	O(e)
普里姆算法	$O(n^2)$	O(n)
克鲁斯卡尔算法	$O(e\log_2 e)$	O(n)
拓扑排序	O(n+e)	O(n)
关键路径	O(n+e)	O(n)
单源最短路径	$O(n^2)$	O(n)
Floyd 算法	$O(n^3)$	$O(n^2)$
折半查找	$O(\log_2 n)$	O(1)
平衡二叉树查找	$O(\log_2 n)$	O(1)
直接插入排序	$O(n^2)$	O(1)
折半插入排序	$O(n^2)$	O(1)
冒泡排序	$O(n^2)$	O(1)
简单选择排序	$O(n^2)$	O(1)
快速排序	最坏 $O(n^2)$、平均 $O(n\log_2 n)$	$O(\log_2 n)$
希尔排序	$O(n^{1+k})$ $(0<k<1)$	O(1)
堆排序	$O(n\log_2 n)$	O(1)
归并排序	$O(n\log_2 n)$	O(n)
基数排序	O(d*(n+r))	O(n+r)

参 考 书 目

[1] 严蔚敏，吴伟民. 数据结构（C 语言版）[M]. 北京：清华大学出版社，2012

[2] 朱昌杰，肖建宇. 数据结构（C 语言版）[M]. 北京：清华大学出版社，2011

[3] 王道论坛. 2019 数据结构考研复习指导[M]. 北京：电子工业出版社，2018

[4] 2014 全国硕士研究生入学统一考试计算机专业基础综合考试大纲解析[M]. 北京：高等教育出版社，2013

[5] 高懿. 2010 年全国硕士研究生入学统一考试计算机学科联考全程辅导[M]. 南京：东南大学出版社，2009

[6] 施游，朱云翔. 全国硕士研究生入学统一考试计算机科学与技术学科联考计算机学科专业基础综合考前串讲[M]. 北京：电子工业出版社，2009

[7] 上海翔高教育，上海恩波学校. 计算机学科专业基础综合（2010 版）[M]. 上海：复旦大学出版社，2009

[8] 朱云翔，胡平. 全国硕士研究生入学统一考试计算机学科专业基础综合冲刺指南[M]. 北京：电子工业出版社，2008